Learning C++ by Building Games with Unreal Engine 4

Second Edition

A beginner's guide to learning 3D game development with C++ and UE4

Sharan Volin

Packt
www.packt.com

Learning C++ by Building Games with Unreal Engine 4
Second Edition

A beginner's guide to learning 3D game development with C++ and UE4

Sharan Volin

Packt

BIRMINGHAM - MUMBAI

Learning C++ by Building Games with Unreal Engine 4
Second Edition

Commissioning Editor: Kunal Chaudhari
Acquisition Editor: Trusha Shriyan
Content Development Editor: Francis Carneiro
Technical Editor: Surabhi Kulkarni
Copy Editor: Safis Editing
Project Coordinator: Pragati Shukla
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Alishon Mendonsa
Production Coordinator: Deepika Naik

First published: February 2015
Second edition: December 2018

Production reference: 1261218

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78847-624-9

`www.packtpub.com`

Contributors

About the author

Sharan Volin has been programming games for more than a decade. In the past, she worked on AAA and other titles for such companies as Sony Online Entertainment, Electronic Arts (Los Angeles), and 7 Studios (Activision). She primarily specialized in UI Programming. She also taught Game Programming for a year at the Art Institute of California. Sharan has both a BS in Computer Science (Games) and an MS in Computer Science from the University of Southern California, as well as degrees and certificates from other schools. Originally from New York, she lived in Los Angeles until she moved to Montreal, Canada with her 3 cats in early 2018 to work as a System Programmer at Behavior Interactive.

I'd like to thank my mother, Linda Volin, an English teacher, who inspired me to read and write. She really supports me in my game programming career. I'd also thank my brother, the rest of my family, my friends, my cats, and the great community of people I've met in the game industry, who made it worth staying in the field even during rough times.

About the reviewer

Agne Skripkaite is a UE4 software engineer with a particular interest in virtual reality applications. Agne has a BSc physics degree with honors from the University of Edinburgh and became a full-time engineer partway through a physics PhD program at Caltech. Over the last few years, Agne has developed for room-scale and seated VR games as part of teams of various sizes, right from two-engineer teams to large development teams. Agne has also served as a user comfort and motion sickness mitigation expert for seated VR applications.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

So, you want to program your own games using **Unreal Engine 4** (**UE4**). You have a great number of reasons to do so: UE4 is powerful—UE4 provides some of the most state-of-the-art, beautiful, and realistic lighting and physics effects available, of the kind used by AAA studios.

UE4 is device-agnostic: code written for UE4 will work on Windows desktop machines, Mac desktop machines, all the major game consoles (if you're an official developer), Android devices, and iOS devices (at the time of writing this book—even more devices may be supported in the future!). So, you can use UE4 to write the main parts of your game once, and after that, deploy to iOS and Android marketplaces without a hitch. (Of course, there will be a few hitches: iOS and Android in-app purchases and notifications will have to be programmed separately, and there could be other differences.)

Who this book is for

This book is for anyone who wants to learn game programming. We'll be going through and creating a simple game, so you will get a good idea of the whole process.

This book is also for anyone who wants to learn C++, particularly C++ 17. We'll be going over the basics of C++ and how to program in it, with an introduction to some of the new features in the latest C++ version.

Finally, the book is for anyone who wants to learn UE4. We'll be using this to create our game. We will be primarily focused on the C++ side, but we will be looking at some basic blueprint development.

What this book covers

`Chapter 1`, *Getting Started with C++ 17*, covers creating your first C++ project in either Visual Studio Community 2017 or Xcode. We'll be creating our first simple C++ program.

`Chapter 2`, *Variables and Memory*, covers different types of variables, the basic method of storing data in C++, as well as pointers, namespaces, and basic input and output in a console application.

`Chapter 3`, *If, Else, and Switch*, covers basic logical statements in C++ that allow you to make choices in your code based on the value in a variable.

`Chapter 4`, *Looping*, covers how to run a piece of code a certain number of times, or until a condition is true. It also covers logical operators, and we'll see our first example of code in UE4.

`Chapter 5`, *Functions and Macros*, covers how we can set up portions of code that can be called from other parts of the code. We also cover how to pass in values or get a return value, and look at some more advanced topics related to variables.

`Chapter 6`, *Objects, Classes, and Inheritance*, covers objects in C++, which are pieces of code that tie data members and member functions together into a bundle of code called a class or struct. We will learn about encapsulation and how it is easier and more efficient to program objects such that they maintain their own internal state.

`Chapter 7`, *Dynamic Memory Allocation*, looks at dynamic memory allocation and how to create space in memory for groups of objects. This chapter introduces you to C- and C++-style arrays and vectors. In most UE4 code, you will use the UE4 editor built-in collection classes.

`Chapter 8`, *Actors and Pawns*, goes into how to create a character and display it on the screen, control your character with axis bindings, and create and display NPCs that can post messages to the HUD.

`Chapter 9`, *Templates and Commonly Used Containers*, goes over how to use templates in C++, and talks about template-based data structures available both in UE4 and in the C++ Standard Template Library.

`Chapter 10`, *Inventory System and Pickup Items*, is where we will code and design a backpack for our player to store items. We will display what the player is carrying in the pack when the user presses the *I* key. We will learn how to set up multiple pickup items for the player.

`Chapter 11`, *Monsters*, looks at how to add a landscape. The player will walk along the path sculpted out for them and then they will encounter an army. You will learn how to instantiate monsters on the screen that run after the player and attack them.

`Chapter 12`, *Building Smarter Monsters with Advanced AI*, covers the basics of AI. We will learn how to use the NavMesh, Behavior Trees, and other AI techniques to make your monsters appear smarter.

`Chapter 13`, *Spell Book*, looks at how to create spells to defend yourself in the game, as well as particle systems to display the spells visually.

`Chapter 14`, *Improving UI Feedback with UMG and Audio*, is about displaying game information to the user with the new UMG UI system. We'll be updating your inventory window to be simpler and much nicer looking using UMG, and I will give you tips on creating your own UIs. It also covers how to add basic audio to enhance your game.

`Chapter 15`, *Virtual Reality and Beyond*, gives an overview of what UE4 is capable of with VR, AR, procedural programming, add-ons, and working with different platforms.

To get the most out of this book

In this book, we don't assume that you have any programming background, so if you're a complete beginner, that's fine! It would help to know your way around a computer, however, as well as knowing some basic game concepts. Of course, if you want to program games, it's very likely you've played at least a few!

We'll be running Unreal Engine 4 and Visual Studio 2017 (or Xcode if you're on a Mac), so you probably want to make sure you're running on a recent, and fairly powerful, computer (and if you want to do VR, make sure your computer is VR-ready).

Also, be ready to work! UE4 uses C++, which you can learn the basics of pretty quickly (and will here), but it can take a long time to really master the language. If you're looking for a quick and easy way to create a game, there are other tools out there, but if you really want to learn skills that could lead to a career programming games, this is a good place to start!

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learning-Cpp-by-Building-Games-with-Unreal-Engine-4-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781788476249_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The first thing we see is an `#include` statement. We are asking C++ to copy and paste the contents of another C++ source file, called `<iostream>`."

A block of code is set as follows:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello, world" << endl;
  cout << "I am now a C++ programmer." << endl;
  return 0;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
string name;
int goldPieces;
float hp;
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Open the **Epic Games Launcher** app. Select **Launch Unreal Engine 4.20.X**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Getting Started with C++17

Academics often describe programming concepts in theory but like to leave implementation to someone else, preferably someone from the industry. In this book, we cover it all: we will describe the theory behind C++ concepts and implement our own game as well. If you're a first-time programmer, you have a lot to learn!

The first thing I will recommend is that you do the exercises. You cannot learn to program simply by reading. You must apply the theory in the exercises to absorb it and be able to use it in the future.

We are going to get started by programming very simple programs in C++. I know that you want to start playing your finished game right now. However, you have to start at the beginning to get to that end (if you really want to, skip over to `Chapter 13`, *Spell Book*, or open some of the samples to get a feel for where we are going).

In this chapter, we will cover the following topics:

- Setting up a new project (in Visual Studio or Xcode)
- Your first C++ project
- How to handle errors
- What are building and compiling?

Setting up our project

Our first C++ program will be written outside of UE4. To start with, I will provide steps for both Xcode and Visual Studio 2017, but after this chapter, I will try to talk about just the C++ code without reference to whether you're using Microsoft Windows or macOS.

Using Microsoft Visual Studio on Windows

In this section, we will install an **integrated development environment** (**IDE**) that will allow you to edit code for Windows, Microsoft's Visual Studio. Please skip to the next section if you are using a Mac.

Downloading and Installing Visual Studio

To start, download and install Microsoft Visual Studio Community 2017.

> The Community edition of Visual Studio is the free version of Visual Studio that Microsoft provides on their website. Go to `https://www.visualstudio.com/downloads/` to download and then start the installation process.

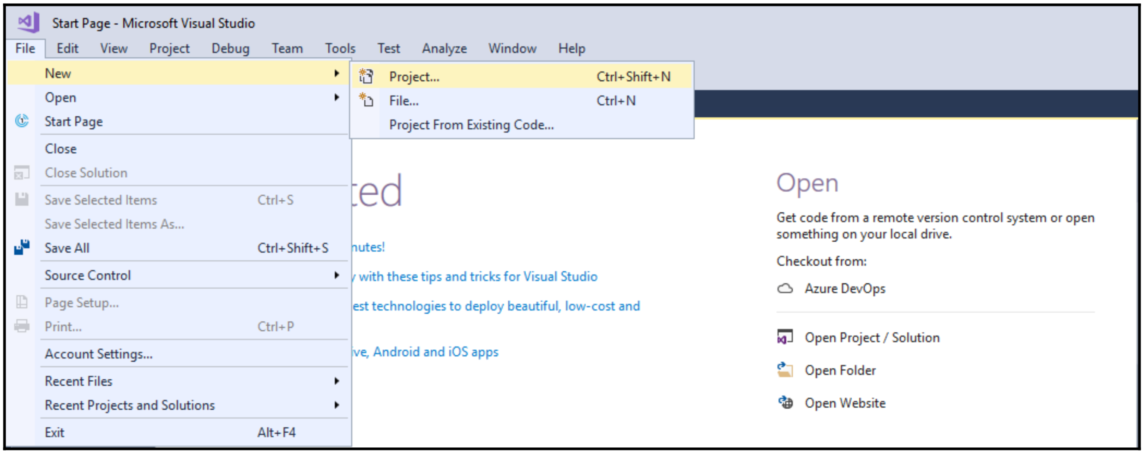
You can find complete installation instructions here: `https://docs.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2017`. When you get to the section on workloads, you will want to choose **Desktop Development with C++**.

Once you have Visual Studio Community 2017, open it. This is how the icon for the software looks:

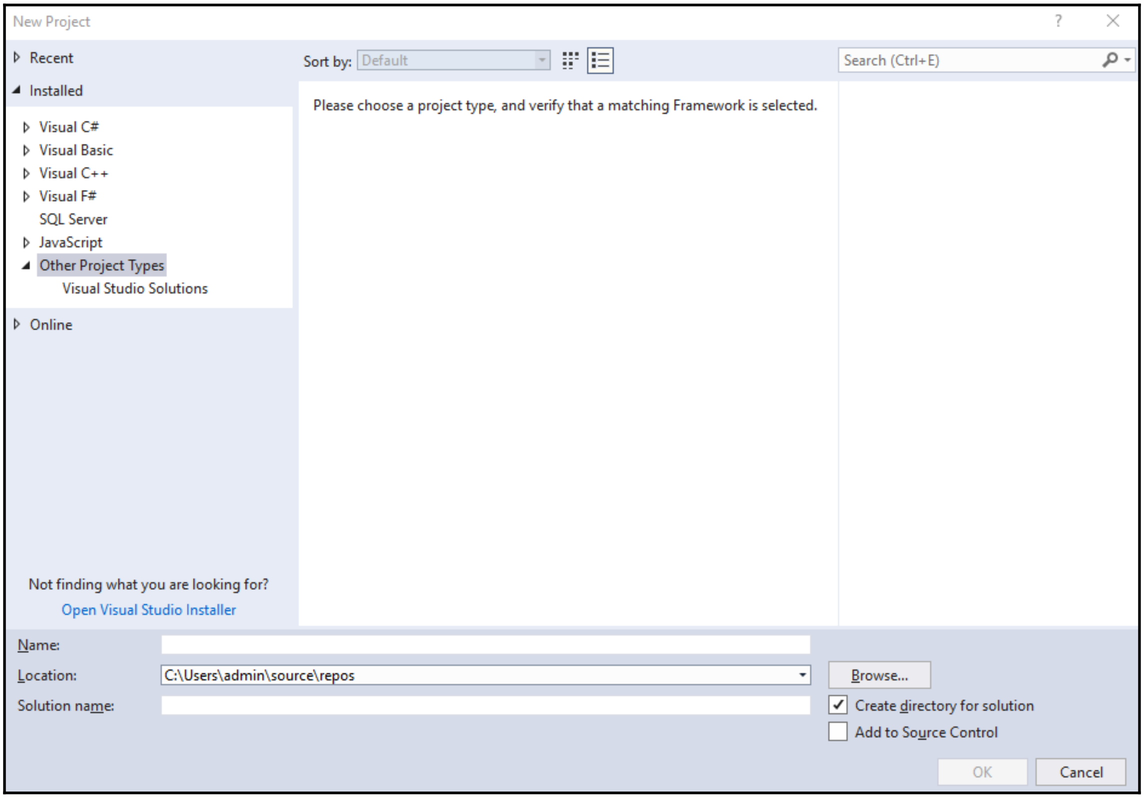
Starting a New Project in Visual Studio

Work through the following steps to get to a point where you can actually type in the code:

1. From the **File** menu, select **New** | **Project...**, as shown in the following screenshot:

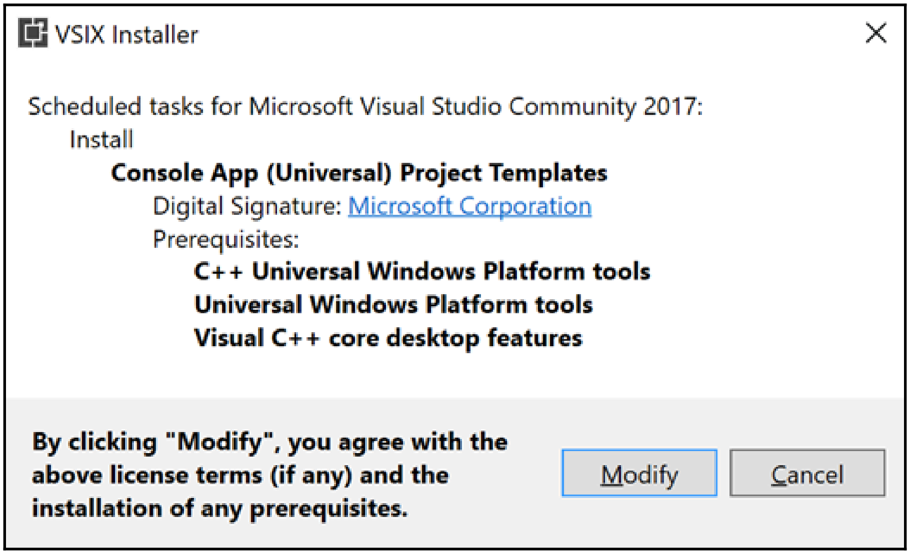


2. You will get the following dialog:

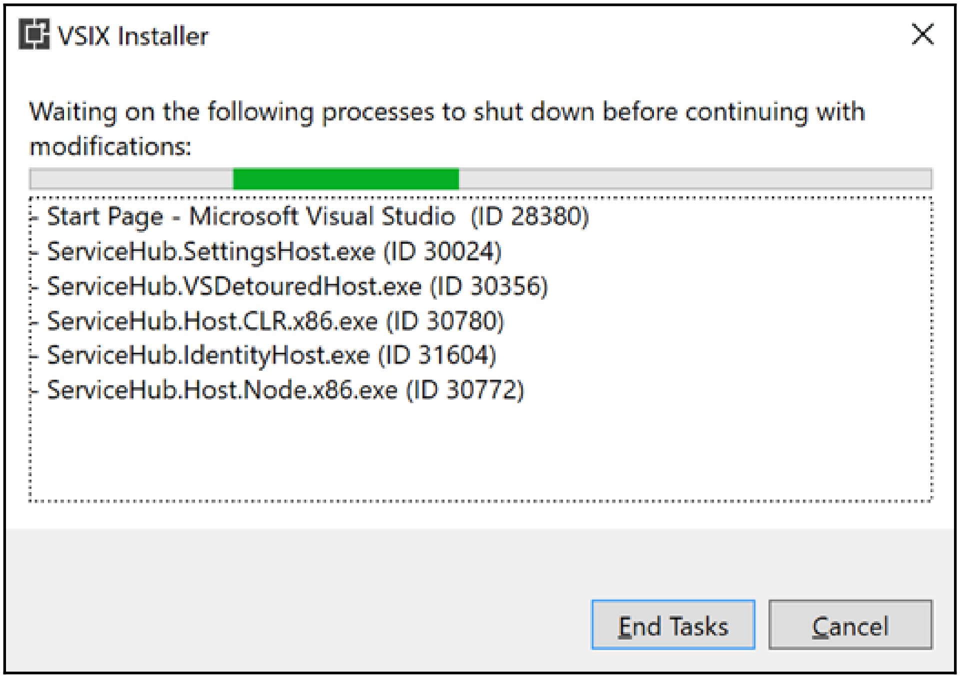


Note that there is a small box at the bottom with the text **Solution name**. In general, **Visual Studio Solutions** might contain many projects. However, this book only works with a single project, but at times you might find it useful to integrate many projects into the same solution.

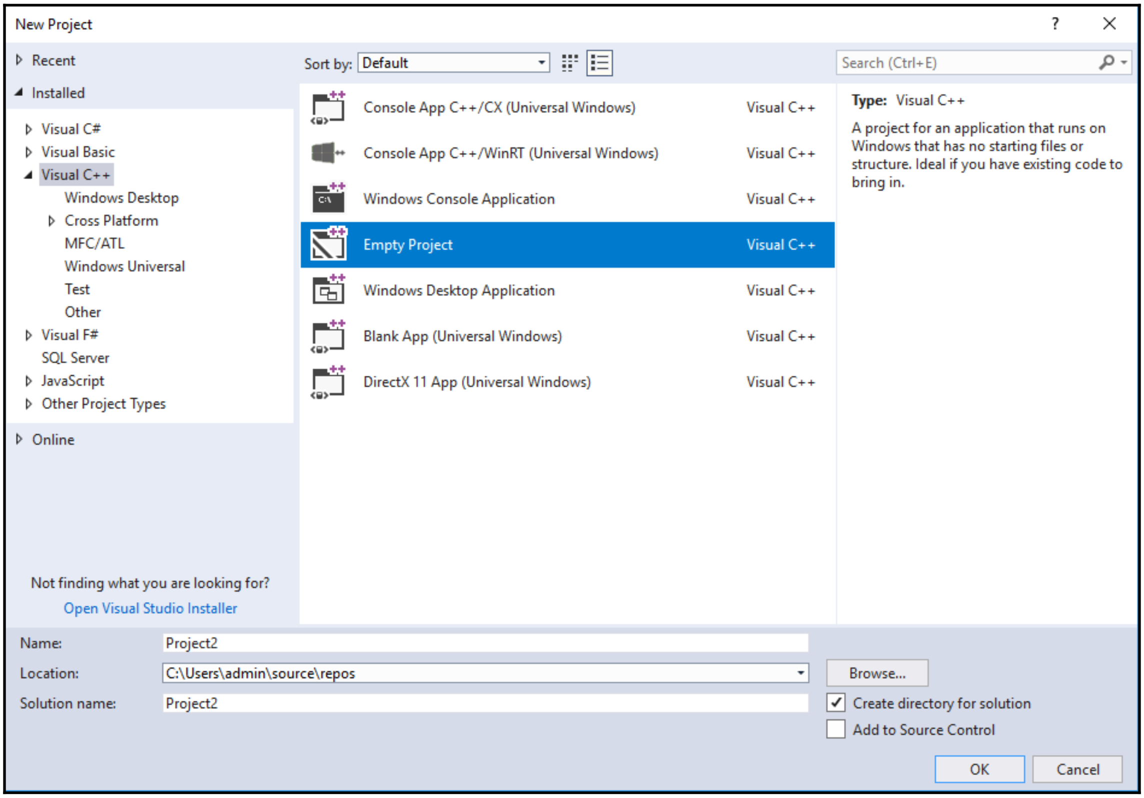
3. There are five things to take care of now, as follows:
    1. Select **Online** | **Templates** | **Visual C++** from the left-hand side panel
    2. Select **Console App (Universal) Project Templates** from the right-hand side panel
    3. Name your app (I used `MyFirstApp`)
    4. Select a folder to save your code
    5. Click on the **OK** button
4. If you have never used this template before, it will open the **VSIX Installer** and show this dialog:



5. Click Modify. It will install and close down Visual Studio. You might need to click End Tasks if you get this dialog:



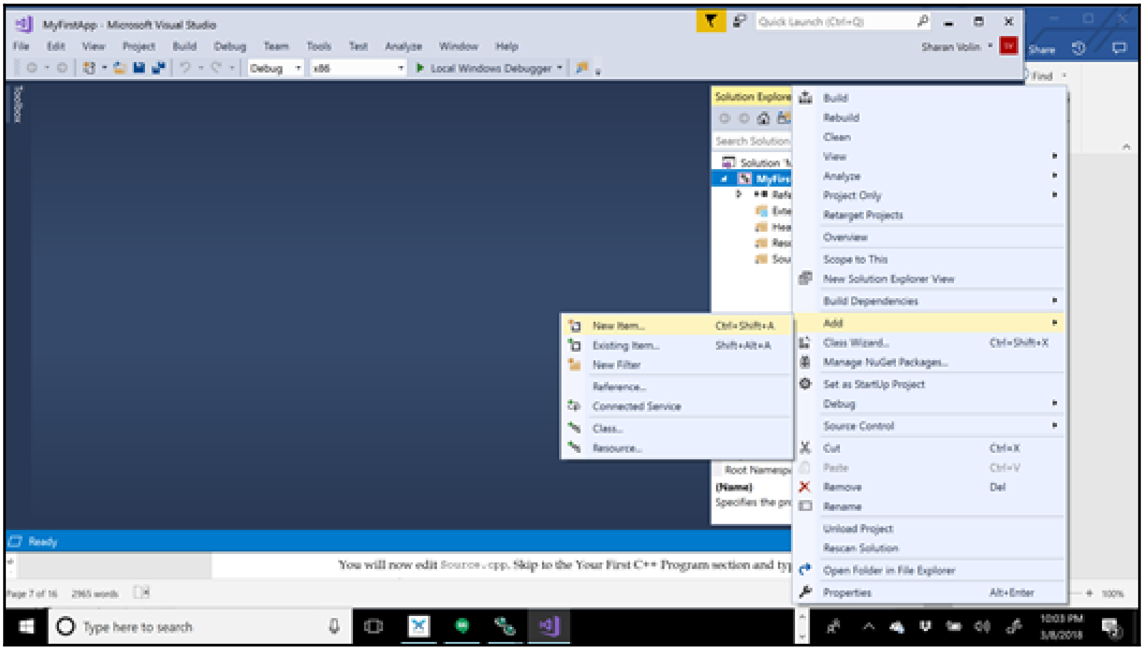
6. Then, it will install the project templates for you. It will take a long time, but you should only need to do this once. When it's done, click **Close** and restart Visual Studio.
7. You need to start over with the previous steps from **File** | **New** | **Project...**. This time, Visual C++ will show up under **Installed**:



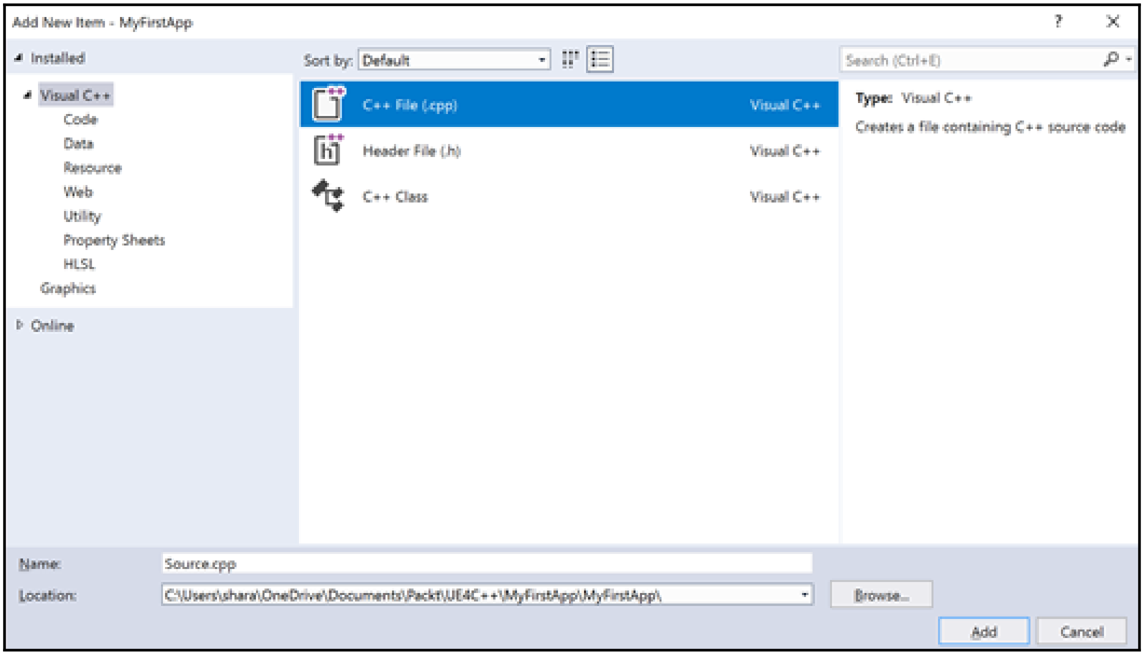
8. Choose Empty Project and you can change the name from Project1 to whatever you want to name it, in my case MyFirstApp.

Now, you are in the Visual Studio 2017 environment. This is the place where you will do all your work and code.

However, we need a file to write our code into. So, we will add a C++ code file to our project by right-clicking on the project name in the Solution Explorer and choosing **Add** | **New Item**, as shown in the following screenshot:



Add your new C++ (`.cpp`) source code file as shown in the following screenshot:



`Source.cpp` is now open and ready for you to add your code. Skip to the *Creating your first C++ program* section and get started.

Using Xcode on a Mac

In this section, we will talk about how to install Xcode on a Mac. Please skip to the next section if you are using Windows.

Downloading and installing Xcode

Xcode is available (for free!) on all Mac computers from the Apple App Store.

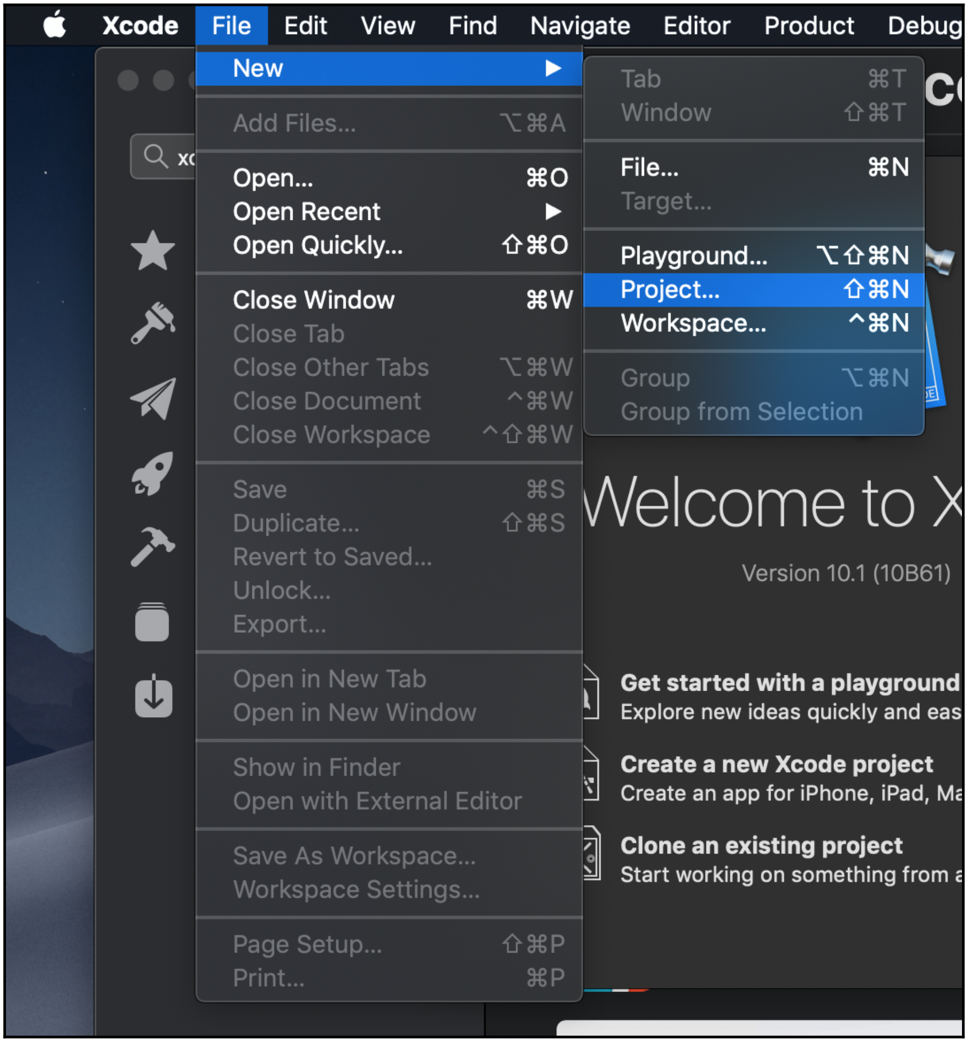
You should get the latest version, if possible. As of this writing it is Xcode 10, but it requires at least macOS Sierra or (preferably) High Sierra. If your Mac is older and running an older operating system, you can download the OS update for free, as long as you're using a machine recent enough to support it.

Just search for Xcode on the Apple App Store, as shown here:

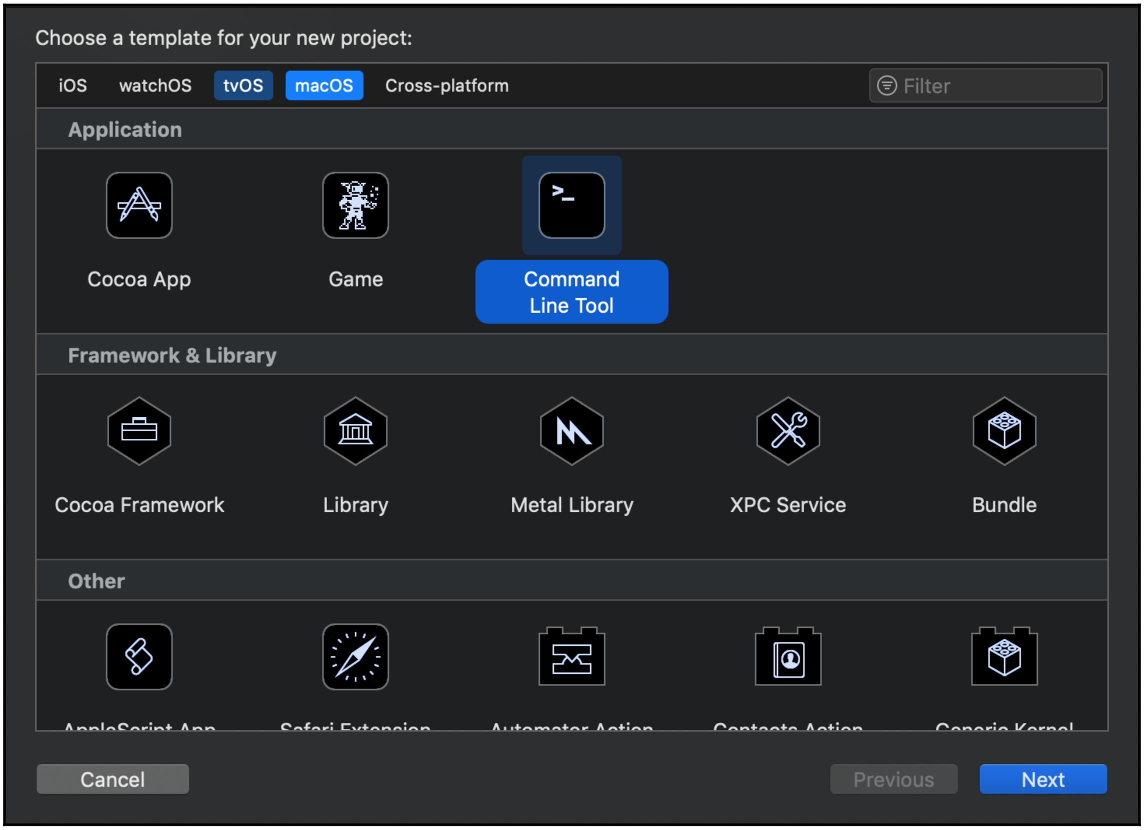
Just click the **Get** button and wait for it to download and install.

Starting a new project in Xcode

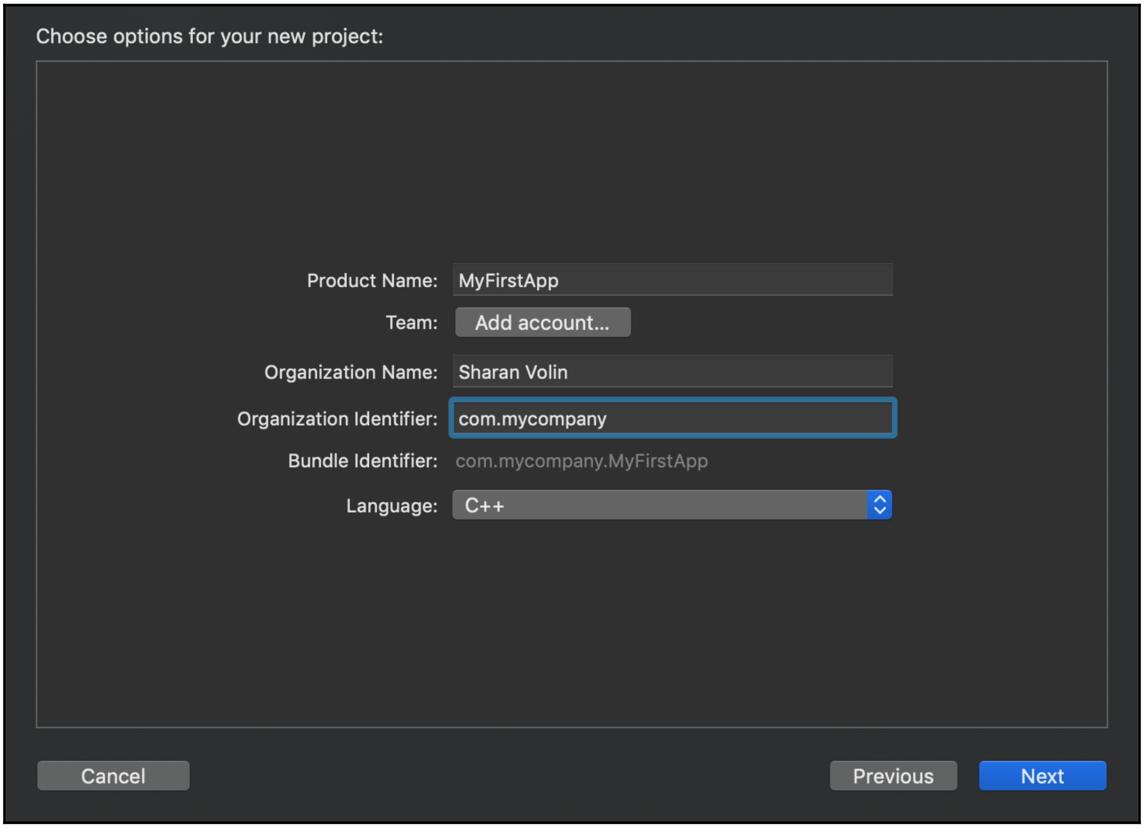
1. Once you have Xcode installed, open it. Then, either choose **Create a new Xcode project** from the opening splash screen or navigate to **File** | **New** | **Project...** from the system's menu bar at the top of your screen, as shown in the following screenshot:



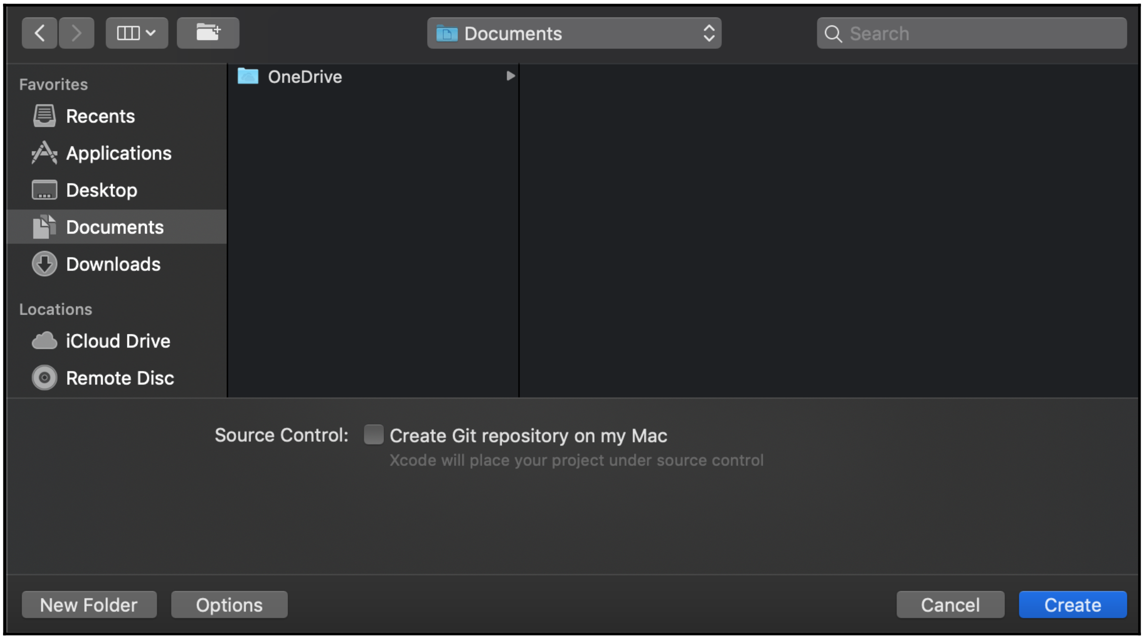
2. In the **New Project** dialog, in the **Application** section under **macOS** at the top of the screen, select **Command Line Tool**. Then, click on **Next**:



3. In the next dialog, name your project. Be sure to fill in all the fields or Xcode won't let you proceed. Make sure that the project's **Type** is set to **C++** and then click on the **Next** button, as shown here:

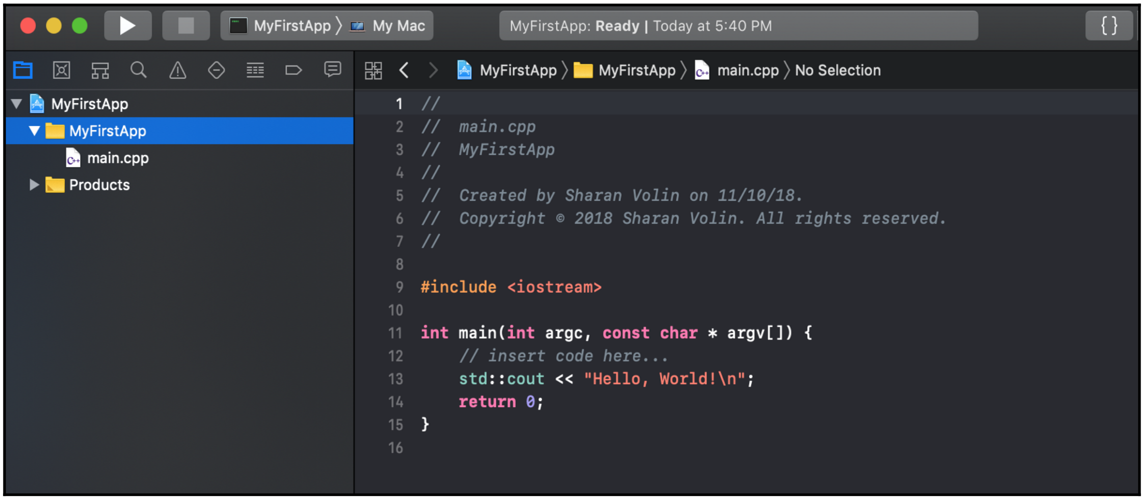


4. The next popup will ask you to choose a location in order to save your project. Pick a spot on your hard drive and save it there. Xcode, by default, creates a Git repository for every project you create. You can uncheck **Create git repository** as we won't cover Git in this chapter, as shown in the following screenshot:



Git is a **version control system**. This basically means that Git takes and keeps snapshots of all the code in your project every so often (every time you *commit* to the repository). Other popular **source control management** (**SCM**) tools are Mercurial, Perforce, and Subversion. When multiple people are collaborating on the same project, the scm tool has the ability to automatically merge and copy other people's changes from the repository to your local code base.

Okay! You are all set up. Click on the `main.cpp` file in the left-hand side panel of Xcode. If the file doesn't appear, ensure that the folder icon at the top of the left-hand side panel is selected first, as shown in the following screenshot:



Creating your first C++ program

We are now going to write some C++ source code. There is a very good reason why we are calling it the source code: it is the source from which the binary executable code is built. The same C++ source code can be built on different platforms such as Mac, Windows, and mobile platforms, and in theory executable code doing the exact same things on each respective platform should result.

In the not-so-distant past, before the introduction of C and C++, programmers wrote code for each specific machine they were targeting individually. They wrote code in a language called assembly language. But now, with C and C++ available, a programmer only has to write code once, and it can be deployed to a number of different machines simply by using different compilers to build the same source code.

TIP

In practice, there are some differences between Visual Studio's flavor of C++ and Xcode's flavor of C++, but these differences mostly appear when working with advanced C++ concepts, such as templates. UE4 is very helpful when working with multiple platforms.

Epic Games put in a lot of work in order to get the same code to work on both Windows and Mac, along with many other platforms such as mobile and game consoles.

A real-world tip
It is important for the code to run in the same way on all machines, especially for networked games or games that allow things such as shareable replays. This can be achieved using standards. For example, the IEEE floating-point standard is used to implement decimal math on all C++ compilers. This means that the result of computations such as 200 * 3.14159 should be the same on all machines. Without standards, different compilers might (for example) round numbers differently, and where there are many calculations and the code needs to be precise, this could cause unacceptable differences.

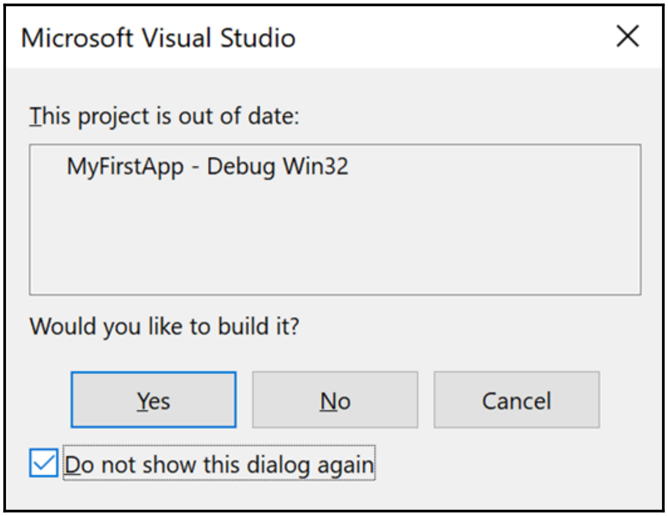
Write the following code in Microsoft Visual Studio or in Xcode:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello, world" << endl;
  cout << "I am now a C++ programmer." << endl;
  return 0;
}
```

To explain what's going on, here is the same code but with comments added (anything after `//` on the same line will be ignored by the compiler, but can help explain what is going on).

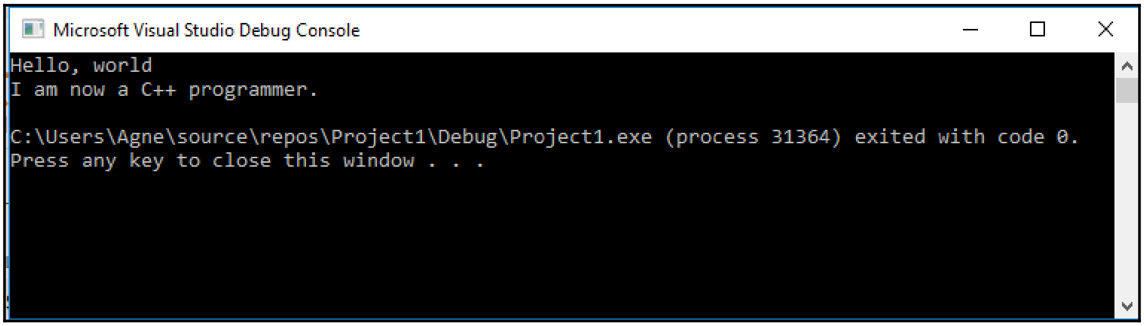
```
#include <iostream>  // Import the input-output library
using namespace std; // allows us to write cout
                     // instead of std::cout
int main()
{
  cout << "Hello, world" << endl;
  cout << "I am now a C++ programmer." << endl;
  return 0;      // "return" to the operating sys
}
```

Press *Ctrl* + *F5* (or use the **Debug** | **Start Without Debugging** menu) to run the preceding code in Visual Studio, or press *Command* + *R* (**Product** | **Run**) to run in Xcode. The first time you press *Ctrl* + *F5* in Visual Studio, you will see this dialog:

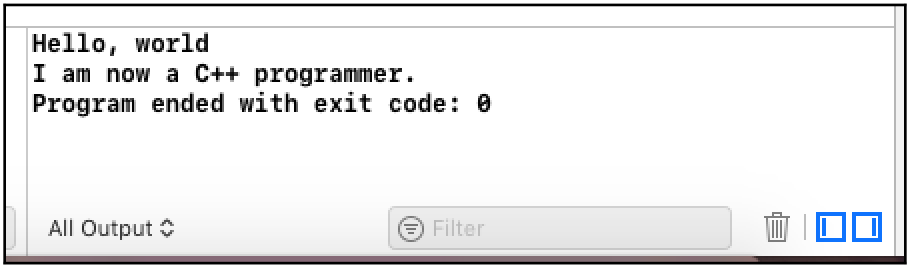


Select **Yes** and **Do not show this dialog again** if you don't want to see this every time you run the program.

Here is what you should see in Windows:



and here it is on Mac:



> **TIP:** If you're on Windows, you will probably notice that the window closes automatically when you run it so you can't see the results. There are various ways around this, including changing the settings to pause and make you press a key to continue. You can get more information here: `https://stackoverflow.com/questions/454681/how-to-keep-the-console-window-open-in-visual-c/1152873#1152873`

The first thing that might come to your mind is *"my! A whole lot of gibberish!"*

Indeed, you rarely see the use of the hash (#) symbol (unless you use Twitter) and curly brace pairs `{}` in normal English text. However, in C++ code, these strange symbols abound. You just have to get used to them.

So, let's interpret this program, starting from the first line.

This is the first line of the program:

```
#include <iostream>  // Import the input-output library
```

This line has two important points to be noted:

1. The first thing we see is an `#include` statement. We are asking C++ to copy and paste the contents of another C++ source file, called `<iostream>`, directly into our code file. `<iostream>` is a standard C++ library that handles all the code that lets us print text to the screen.
2. The second thing we notice is a `//` comment. As mentioned earlier, C++ ignores any text after a double slash (`//`) until the end of that line. Comments are very useful to add in plain text explanations of what some code does. You might also see `/* */` multiline C-style comments in the source. Surrounding any text (even over multiple lines) in C or C++ with slash-star `/*` and star-slash `*/` gives an instruction to have that code removed by the compiler.

This is the next line of code:

```
using namespace std; // allows us to write cout
                     // instead of std::cout
```

The comments beside this line explain what the `using` statement does: it just lets you use a shorthand (for example, `cout`) instead of the fully qualified name (which, in this case, would be `std::cout`) for a lot of our C++ code commands. Some people don't like a `using namespace std;` statement; they prefer to write the `std::cout` longhand every time they want to use `cout`. You can get into long arguments over things like this. In this section of the text, we prefer the brevity that we get with the `using namespace std;` statement.

Also, note the comments on the second line of this section are lined up with the ones on the previous line. This is good programming practice because it shows visually that it is a continuation of the previous comment.

This is the next line:

```
int main()
```

This is the application's starting point. You can think of `main` as the start line in a race. The `int main()` statement is how your C++ program knows where to start.

If you don't have an `int main()` program marker or if `main` is spelled incorrectly, then your program just won't work because the program won't know where to start.

The next line is a character you don't see often:

```
{
```

This `{` character is not a sideways mustache. It is called a curly brace, and it denotes the starting point of your program.

The next two lines print text to the screen:

```
cout << "Hello, world" << endl;
cout << "I am now a C++ programmer." << endl;
```

The `cout` statement stands for console output. Text between double quotes will get an output to the console exactly as it appears between the quotes. You can write anything you want between double quotes, except a double quote, and it will still be valid code. Also, note that `endl` tells `cout` to add an end line (carriage return) character, which is very useful for formatting.

To enter a double quote between double quotes, you need to stick a backslash () in front of the double quote character that you want inside the string, as shown here:

```
cout << "John shouted into the cave \"Hello!\" The cave
echoed."
```

The `\"` symbol is an example of an escape sequence. There are other escape sequences that you can use; the most common escape sequence you will find is `\n`, which is used to jump the text output to the next line.

The last line of the program is the `return` statement:

```
return 0;
```

This line of code indicates that the C++ program is quitting. You can think of the `return` statement as returning to the operating system.

Finally, the end of your program is denoted by the closing curly brace, which is an opposite-facing sideways mustache:

```
}
```

Semicolons

Semicolons (;) are important in C++ programming. Notice in the preceding code example that most lines of code end in a semicolon. If you don't end each line with a semicolon, your code will not compile, and if that happens, your employer won't be very happy (of course, once you've been doing this for a while you'll find and fix these issues well before they even find out about it).

Handling errors

If you make a mistake while entering code, then you will have a syntax error. In the face of syntax errors, C++ will scream bloody murder and your program will not even compile; also, it will not run.

Let's try to insert a couple of errors into our C++ code from earlier:

```
#include·<iostreams>¶
using·namespace·std;¶
int·maind(¶
{¶
··count·<<·"Hello,·world·<<·ender:¶
··count·<<·"I·am·nowb·a·C++·programmer."·<<·ender;¶

}¶
```

Error

Warning! This code listing contains errors. It is a good exercise to find all the errors and fix them!

As an exercise, try to find and fix all the errors in this program.

Note that if you are extremely new to C++, this might be a hard exercise. However, this will show you how careful you need to be when writing C++ code.

Fixing compilation errors can be a nasty business. However, if you input the text of this program into your code editor and try to compile it, it will cause the compiler to report all the errors to you. Fix the errors, one at a time, and then try to recompile (start with the first one on the list, because it could be causing some of the later errors). A new error will pop up or the program will just work, as shown in the following screenshot:

```
#include <iostreams>
using namespace std;

int maind(
{
   count << "Hello, world << ender:
   count << "I am nowb a C++ programmer." << ender;
}

```

'iostreams' file not found

Your compiler shows you the errors in your code when you try to compile it (although if you're using Visual Studio, it will ask if you want to run the previous successful build first).

The reason I am showing you this sample program is to encourage the following workflow as long as you are new to C++:

1. Always start with a working C++ code example. You can fork off a bunch of new C++ programs from the *Creating your first C++ program* section.
2. Make your code modifications in small steps. When you are new, compile after writing each new line of code. Do not code for one to two hours and then compile all that new code at once.
3. You can expect it to be a couple of months before you can write code that performs as expected the first time you write it. Don't get discouraged. Learning to code is fun.

Warnings in C++

The compiler will flag things that it thinks might be mistakes. These are another class of compiler notices known as warnings. Warnings are problems in your code that you do not have to fix for your code to run but are simply recommended to be fixed by the compiler. Warnings are often indications of code that is not quite perfect, and fixing warnings in code is generally considered good practice.

However, not all warnings are going to cause problems in your code. Some programmers prefer to disable the warnings that they do not consider to be an issue (for example, warning 4018 warns against asigned/unsigned mismatch, which you will most likely see later).

What is building and compiling?

You might have heard of a computer process term called compiling. Compiling is the process of converting your C++ program into code that can run on a CPU. Building your source code means the same thing as compiling it.

See, your source `code.cpp` file will not actually run on a computer. It has to be compiled first for it to run.

This is the whole point of using Microsoft Visual Studio Community or Xcode. Visual Studio and Xcode are both compilers. You can write C++ source code in any text editing program—even in Notepad. But you need a compiler to run it on your machine.

Every operating system typically has one or more C++ compilers that can compile C++ code to run on that platform. On Windows, you have Visual Studio and Intel C++ Studio compiler. On Mac, there is Xcode, and on all of Windows, Mac, and Linux, there is the **GNU Compiler Collection** (GCC).

The same C++ code that we write (source) can be compiled using different compilers for different operating systems, and in theory they should produce the same result. The ability to compile the same code on different platforms is called portability. In general, portability is a good thing.

Example output

Here is the screenshot is of your first C++ program:

```cpp
#include <iostream>  // Import the input-output library
using namespace std; // allows us to write cout
                     // instead of std::cout

int main()
{
    cout << "Hello, world" << endl;
    cout << "I am now a C++ programmer." << endl;
    return 0;
}
```

And the following screenshot is of its output, your first victory:

> There is another class of programming languages called scripting languages. These include languages such as PHP, Python, and `ActionScript`. Scripted languages are not compiled; for JavaScript, PHP, and ActionScript, there is no compilation step. Rather, they are interpreted from the source as the program is run. The good thing about scripting languages is that they are usually platform-independent from the word go, because interpreters are very carefully designed to be platform-independent.

Exercise - ASCII art

Game programmers love ASCII art. You can draw a picture using only characters. Here's an example of an ASCII art maze:

```
cout << "****************" << endl;
cout << "*............*.*" << endl;
cout << "*.*.*******..*.*" << endl;
cout << "*.*.*..........*" << endl;
cout << "*.*.*.**********" << endl;
cout << "***.***........*" << endl;
```

Construct your own maze in C++ code or draw a picture using characters.

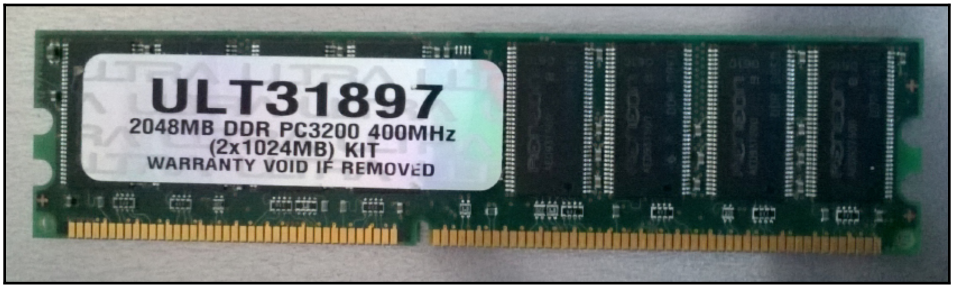
Summary

To sum up, we learned how to write our first program in the C++ programming language in our integrated development environment (IDE, Visual Studio, or Xcode). This was a simple program, but you should count getting your first program to compile and run as your first victory. In the upcoming chapters, we'll put together more complex programs and start using Unreal Engine for our games.

2
Variables and Memory

To write your C++ game program, you will need your computer to remember a lot of things, such as where in the world the player is, how many hit points they have, how much ammunition they have left, where the items are in the world, what power-ups they provide, and the letters that make up the player's screen name.

The computer that you have actually has a sort of electronic sketchpad inside it called **memory**, or RAM. Physically, computer memory is made out of silicon and it looks similar to what is shown in the following photograph:



Does this RAM look like a parking garage? Because that's the metaphor we're going to use.

RAM is short for random access memory. It is called random access because you can access any part of it at any time. If you still have some CDs lying around, they are an example of non-random access. CDs are meant to be read and played back in order. I still remember jumping tracks on Michael Jackson's *Dangerous* album way back when switching tracks on a CD took a lot of time! Hopping around and accessing different cells of RAM, however, doesn't take much time at all. RAM is a type of fast memory access known as flash memory.

RAM is called volatile flash memory because when the computer is shut down, the RAM's contents are cleared, and the old contents of the RAM are lost unless they were saved to the hard disk first.

For permanent storage, you have to save your data to a hard disk. There are two main types of hard disks:

- Platter-based **hard disk drives** (**HDDs**)
- **Solid-state drives** (**SSDs**)

SSDs are more modern than platter-based HDDs, since they use RAM's fast access (flash) memory principle. Unlike RAM, however, the data on an SSD persists after the computer is shut down. If you can get an SSD, I'd highly recommend that you use it! Platter-based drives are outdated.

While a program is running, it is much faster to access data stored in the RAM than it is to access it from either HDDs or SSDs, so we need a way to reserve a space on the RAM and read and write from it. Fortunately, C++ makes this easy.

Variables

A saved location in computer memory that we can read or write to is called a **variable**.

A variable is a component whose value can vary. In a computer program, you can think of a variable as a container in which you can store some data. In C++, these data containers (variables) have types, and names you can use to refer to them. You have to use the right type of data container to save your data in your program.

If you want to save an integer, such as 1, 0, or 20, you will use an `int` type container. You can use float-type containers to carry around floating-point (decimal) values, such as 38.87, and you can use string variables to carry around strings of letters (think of it as a *string of pearls*, where each letter is a pearl).

You can think of your reserved spot in RAM like reserving a parking space in a parking garage: once we declare our variable and get a spot for it, no one else (not even other programs running on the same machine) will be given that piece of RAM by the operating system. The RAM beside your variable might be unused or it might be used by other programs.

TIP

The operating system exists to keep programs from stepping on each other's toes and accessing the same bits of computer hardware at the same time. In general, civil computer programs should not read or write to each other's memory. However, some types of cheat programs (for example, maphacks) secretly access your program's memory. Programs such as PunkBuster were introduced to prevent cheating in online games.

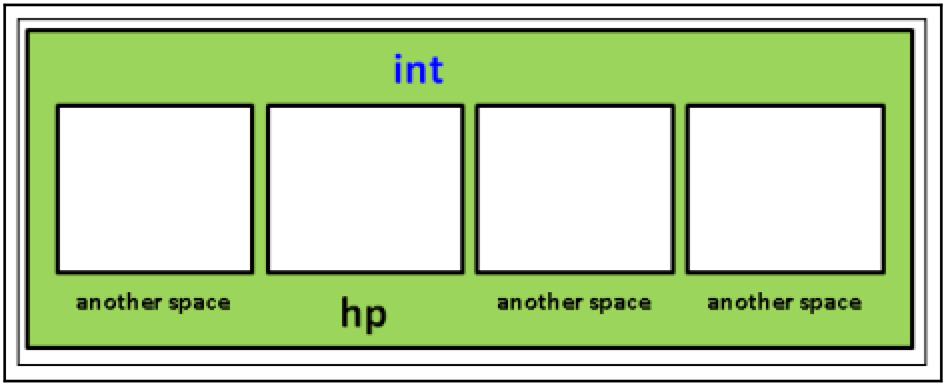
Declaring variables – touching the silicon

Reserving a spot in computer memory using C++ is easy. We want to name our chunk of memory that we will store our data in with a good, descriptive name.

For example, say we know that player **hit points** (**hp**) will be an integer (whole) number, such as 1, 2, 3, or 100. To get a piece of silicon to store the player's `hp` in memory, we will declare the following line of code:

```
int hp;     // declare variable to store the player's hp
```

This line of code reserves a small chunk of RAM to store an integer (`int` is short for integer) called `hp`. The following is an example of our chunk of RAM used to store the player's `hp`. This reserves a parking space for us in the memory (among all the other parking spaces), and we can refer to this space in memory by its label (`hp`):



Among all the other spaces in memory, we get one spot to store our hp data.

> TIP: When you name a variable, there are a few rules. Variable names can't start with a number, and there are certain "reserved words" the compiler won't let you use (usually because they are used by C++ itself). You will learn these as you learn more C++, or you can look for lists of reserved words online.

Notice how the variable space is type-marked in this diagram as `int` if it is a space for a double or a different type of variable. C++ remembers the spaces that you reserve for your program in memory not only by name, but by the type of variable it is as well.

Notice that we haven't put anything in hp's box yet! We'll do that later—right now, the value of the `hp` variable is not set, so it will have the value that was left in that parking space by the previous occupant (the value left behind by another program, perhaps). Telling C++ the type of the variable is important! Later, we will declare a variable to store decimal values, such as 3.75.

Reading and writing to your reserved spot in memory

Writing a value into memory is easy! Once you have an `hp` variable, you just write to it using the = sign:

```
hp = 500;
```

Voila! The player has 500 hp.

Reading the variable is equally simple. To print out the value of the variable, simply put the following:

```
cout << hp << endl;
```

This will print the value stored inside the `hp` variable. The `cout` object is smart enough to figure out what type of variable it is and print the contents. If you change the value of `hp` and then use `cout` again, the most up-to-date value will be printed, as shown here:

```
hp = 1200;
cout << hp << endl; // now shows 1200
```

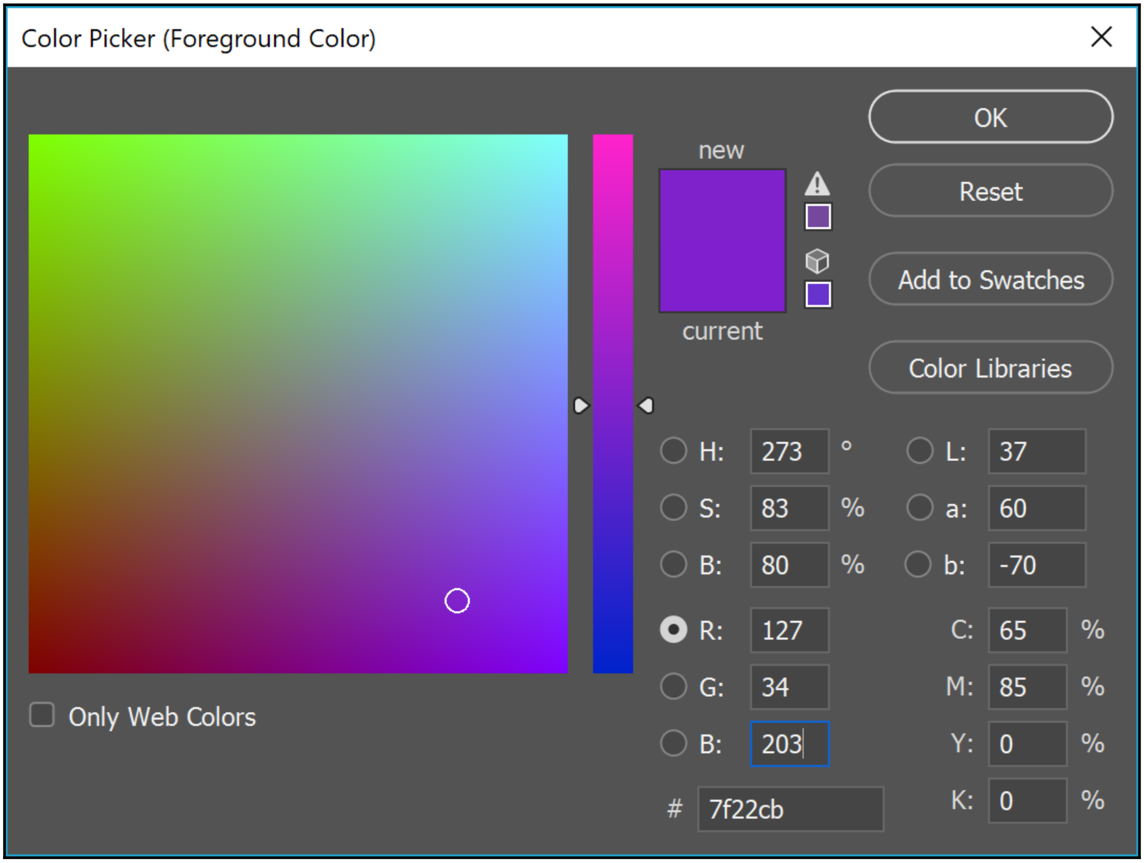
Numbers and math

The heading says it all; in this section, we'll dive into the importance of numbers and math in C++.

Numbers are everything

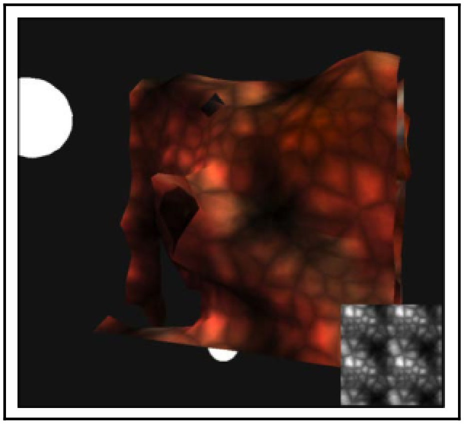
Something that you need to get used to when you start computer programming is that a surprising number of things can be stored in computer memory as just numbers. A player's hp? As we just saw in the previous section, hp can just be an integer number. If the player gets hurt, we reduce this number. If the player gains health, we increase the number.

Colors can be stored as numbers too! If you've used standard image editing programs, there may be sliders that indicate color as how much red, green, and blue are being used, such as Pixelmator's color sliders, if you've used that. Photoshop doesn't have sliders but does show you the numbers and allow you to edit them directly to change the color. A color is then represented by three numbers. The purple color shown in the following screenshot is (**R**: `127`, **G**: `34`, **B**: `203`):



As you can see, Photoshop allows you to use other numbers to represent colors, such as HSB (hue, saturation, brightness), an alternate way of representing color, or CMYK (cyan, magenta, yellow, black) which is used for printing, since professional printing presses use those color inks in printing. For viewing on a computer monitor, you will generally stick to RGB color representation since that is what monitors use.

What about world geometry? These are also just numbers; all we have to do is store a list of 3D space points (*x*, *y*, and *z* coordinates) and then store another list of points that explain how those points can be connected to form triangles. In the following screenshot, we can see how 3D space points are used to represent world geometry:



The combination of numbers for colors and numbers for 3D space points will let you draw large and colored landscapes in your game world.

The trick with the preceding examples is how we interpret the stored numbers so that we can make them mean what we want them to mean.

More on variables

You can think of variables as pet carriers. A cat carrier can be used to carry a cat but not a dog. Similarly, you should use a float-type variable to carry decimal-valued numbers. If you store a decimal value inside an `int` variable, it will not fit:

```
int x = 38.87f;
cout << x << endl; // prints 38, not 38.87
```

What's really happening here is that C++ does an automatic type conversion on `38.87`, *transmogrifying* it to an integer to fit in the `int` carrying case. It drops the decimal to convert `38.87` into the integer value `38`.

So, for example, we can modify the code to include the use of three types of variables, as shown in the following code:

```
#include <iostream>
#include <string>  // need this to use string variables!
using namespace std;
int main()
{
  string name;
  int goldPieces;
  float hp;
  name = "William"; // That's my name
  goldPieces = 322; // start with this much gold
  hp = 75.5f;       // hit points are decimal valued
  cout << "Character " << name << " has "
           << hp << " hp and "
           << goldPieces << " gold.";
}
```

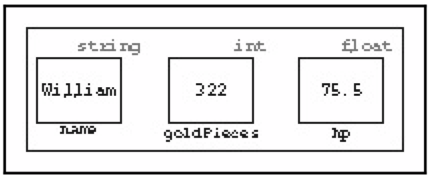
In the first three lines, we declare three boxes to store our data parts in, as shown here:

```
string name; int goldPieces; float hp;
```

These three lines reserve three spots in memory (like parking spaces). The next three lines fill the variables with the values we desire, as follows:

```
name = "William";
goldPieces = 322;
hp = 75.5f;
```

In computer memory, this will look like the following diagram:



You can change the contents of a variable at any time. You can write a variable using the = assignment operator, as follows:

```
goldPieces = 522;// = is called the "assignment operator"
```

You can also read the contents of a variable at any time. That's what the next three lines of code do, as shown here:

```
cout << "Character " << name << " has "
     << hp << " hp and "
     << goldPieces << " gold.";
```

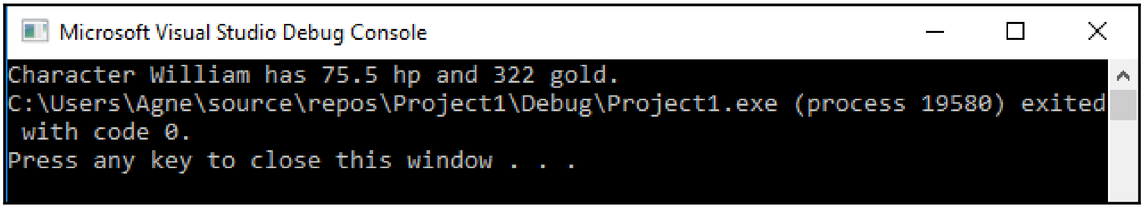
Take a look at the following line:

```
cout << "I have " << hp << " hp." << endl;
```

There are two uses of the word `hp` in this line. One is between double quotes, while the other is not. Words between double quotes are always output exactly as you typed them. When double quotes are not used (for example, `<< hp <<`), a variable lookup is performed. If the variable does not exist, then you will get a compiler error (undeclared identifier).

There is a space in memory that is allocated for the name, a space for how many `goldPieces` the player has, and a space for the hp of the player.

This is what you should see when you run the program:



TIP: In general, you should always try to store the right type of data inside the right type of variable. If you happen to store the wrong type of data, your code may misbehave. For example, accidentally storing a float into an `int` variable will make you lose the decimal points, and storing the value of a char in an `int` will give you the ASCII value, but will no longer treat it as a letter. Sometimes, it even doesn't have any type of automatic type conversion so it won't know how to handle the value at all.

Math in C++

Math in C++ is easy to do; `+` (plus), `-` (minus), `*` (times), and `/` (divide by) are all common C++ operations, and the proper **brackets**, **exponents**, **division**, **multiplication**, **addition**, and **subtraction** (**BEDMAS**) order will be followed. For example, we can do as shown in the following code:

```
int answer = 277 + 5 * 4 / 2 + 20;
```

Of course, if you want to be absolutely sure of the order, it is always a good idea to use parentheses. Another operator that you might not be familiar with yet is % (modulus). Modulus (for example, 10 % 3) finds the remainder when `x` (10) is divided by `y` (3). See the following table for examples:

Operator (name)	Example	Answer
+ (plus)	7 + 3	10
- (minus)	8 - 5	3
* (times)	5*6	30
/ (division)	12/6	2
% (modulus)	10 % 3	1 (because 10/3 is 3 and the remainder = 1).

However, we often don't want to do math in this manner. Instead, we usually want to change the value of a variable by a certain computed amount. This is a concept that is harder to understand. Say the player encounters an imp and is dealt 15 damage.

The following line of code will be used to reduce the player's `hp` by `15` (believe it or not):

```
hp = hp - 15;                    // probably confusing :)
```

You might ask why. Because on the right-hand side, we are computing a new value for hp (`hp-15`). After the new value for hp is found (15 less than what it was before), the new value is written into the `hp` variable.

Think of `hp` as a painting at a specific spot on a wall. `-15` tells you to draw a mustache on the painting but leave it in the same place. The new, mustachioed painting is now `hp`.

Pitfall
An uninitialized variable has the bit pattern that was held in memory for it before. Declaring a variable does not clear the memory.

So, say we used the following line of code:

```
int hp;
hp = hp - 15;
```

The second line of code reduces the hp by 15 from its previous value. What was its previous value if we never set `hp = 100` or so? It could be 0, but not always.
One of the most common errors is to proceed with using a variable without initializing it first.

The following is a shorthand syntax for doing this:

```
hp -= 15;
```

Besides `-=`, you can use `+=` to add an amount to a variable, `*=` to multiply a variable by an amount, and `/=` to divide a variable by an amount.

If you're using an `int` and want to increment (or decrement) it by 1, you can shorten the syntax. You don't need to write the following:

```
hp = hp + 1;
hp = hp - 1;
```

You can instead do any of the following:

```
hp++;
++hp;
hp--;
--hp;
```

Putting it before the variable increments or decrements it before the value is used (if you're using it in a larger statement). Putting it after updates the variable after it is used.

Exercises

Write down the value of `x` after performing the following operations, then check with your compiler:

Exercises	Solutions
`int x = 4; x += 4;`	8
`int x = 9; x-=2;`	7
`int x = 900; x/=2;`	450
`int x = 50; x*=2;`	100
`int x = 1; x += 1;`	2
`int x = 2; x -= 200;`	-198
`int x = 5; x*=5;`	25

Generalized variable syntax

In the previous section, you learned that every piece of data that you save in C++ has a type. All variables are created in the same way; in C++, variable declarations are of the following form:

```
variableType variableName;
```

The `variableType` object tells you what type of data we are going to store in our variable. The `variableName` object is the symbol we'll use to read or write to that piece of memory.

Primitive types

We previously talked about how all the data inside a computer will at some point be a number. Your computer code is responsible for interpreting that number correctly.

It is said that C++ only defines a few basic data types, as shown in the following table:

`Char`	A single letter, such as *a*, *b*, or +. It is stored as a number value from -127 to 127 using ASCII, a standard that assigns specific number values to each character.
`Short`	An integer from `-32,767` to `+32,768`.
`Int`	An integer from `-2,147,483,647` to `+2,147,483,648`.
`Long`	An integer from `-2,147,483,647` to `+2,147,483,648`.
`Float`	Any decimal value from approx. $-1x10^{38}$ to $1x10^{38}$.
`Double`	Any decimal value from approx. $-1x10^{308}$ to $1x10^{308}$.
`Bool`	True or false.

There are unsigned versions of each of the variable types mentioned in the preceding table (except of course Bool, which wouldn't really make sense). An unsigned variable can contain natural numbers, including 0 (x >= 0). An unsigned `short`, for example, might have a value between `0` and `65535`. You can also get even bigger integers if necessary using `long long` or `long long int`.

> The size of variables can sometimes be different for different compilers, or depending on whether you are compiling for a 32-bit or 64-bit operating system. Keep that in mind if you find yourself working on something different in the future.

In this case, we are focusing on Visual Studio or Xcode and (most likely) 64-bit.

> If you're interested in the difference between float and double, please feel free to look it up on the internet. I will keep my explanations only for the most important C++ concepts used for games. If you are curious about something that's not covered by this text, feel free to look it up.

Advanced variable topics

Newer versions of C++ have added a few new features related to variables, and there are others that haven't been mentioned yet. Here are a few things you should keep in mind.

Automatically detecting type

Starting with C++ 11, there is a new variable *type* you can use for cases where you may not be sure what type you are expecting to get. This new type is called `auto`. What it means is that it will detect the type of whatever value you first assign to it and then use that. Say you type the following:

```
auto x = 1.5;
auto y = true;
```

If you do this, `x` will automatically be a float and `y` will be a Boolean. In general, you want to use the actual variable type if you know it (and most of the time you will), and as a beginner it is better to avoid using it. You should be able to recognize it when you see it, however, and know about it if you do run into a case where you need it eventually.

Enums

Enums have existed for a long time, but starting with C++ 11 you have more control over them. The idea behind an enum is sometimes you want to track different types of things in a game, and you just want an easy way to give each a value that tells you what it is, and that you can later check. An enum looks like the following:

```
enum weapon {
    sword = 0;
    knife,
    axe,
    mace,
    numberOfWeaponTypes,
    defaultWeapon = mace
}; // Note the semicolon at the end
```

This creates each of these weapon types and assigns each a unique value by adding 1 to each, so knife would be equal to 1, axe to 2, and so on. Note that you do not need to set the first one to 0 (it does that automatically) but if you want to start with a different number you can do that (and it's not just the first one that can be set to a specific value). You can also assign any `enum` member to a different one and it will have the same value (in this example, `defaultWeapon` has the same value as `mace`: 3). Any time you assign a specific value anywhere in the enum list, any types you add after that on the list will start going up by 1 from that value.

Enums have always contained an int value, but starting with C++ 11 you have the ability to specify a variable type. For example, you might want to do something like the following:

```
enum isAlive : bool {
    alive = true,
    dead = false
}
```

While you could do this with 0 and 1, you might find this more convenient in some cases.

const variables

Sometimes you will have a value you do not want to change ever during the game. You don't want things such as the number of lives, your maximum hp, the amount of xp you need to reach a specific level, or your movement speed to change (unless your character does reach that level, in which case you might switch to a different constant value).

In some cases, an `enum` will work for this, but for single values it's easier to create a new variable and declare it to be `const`. Here's an example:

```
const int kNumLives = 5;
```

Putting `const` in front of the variable type tells the program to never allow that value to be changed, and if you try it will give you an error. Putting `k` in front of the variable name is a common naming convention for `const` variables. Many companies will insist that you follow that standard.

Building more complex types

It turns out that these simple data types alone can be used to construct arbitrarily complex programs. *How?* you ask. Isn't it hard to build a 3D game using just floats and integers?

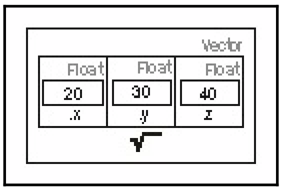
It is not really difficult to build a game from `float` and `int`, but more complex data types help. It would be tedious and messy to program if we used loose floats for the player's position.

Object types – struct

C++ gives you structures to group variables together, which will make your life a lot easier. Take the example of the following block of code:

```
#include <iostream>
using namespace std;
struct Vector        // BEGIN Vector OBJECT DEFINITION
{
  float x, y, z;     // x, y and z positions all floats
};                   // END Vector OBJECT DEFINITION.
// The computer now knows what a Vector is
// So we can create one.
int main()
{
  Vector v; // Create a Vector instance called v
  v.x=20, v.y=30, v.z=40; // assign some values
  cout << "A 3-space vector at " << v.x << ", " << v.y << ", " <<
   v.z << endl;
}
```

The way this looks in memory is pretty intuitive; a **Vector** is just a chunk of memory with three floats, as shown in the following diagram:



Don't confuse the `struct Vector` in the preceding screenshot with the `std::vector` of the **Standard Template Library** (**STL**)—we'll get into that later. The preceding `Vector` object is meant to represent a three-space vector, while the STL's `std::vector` type represents a collection of values.

Here are a couple of review notes about the preceding code listing.

First, even before we use our `Vector` object type, we have to define it. C++ does not come with built-in types for math vectors (it only supports scalar numbers, and they thought that was enough!). So, C++ lets you build your own object constructions to make your life easier. We first had the following definition:

```
struct Vector        // BEGIN Vector STRUCT DEFINITION
{
  float x, y, z;     // x, y, and z positions all floats
};                   // END Vector STRUCT DEFINITION.
```

This tells the computer what a `Vector` is (it's three floats, all of which are declared to be sitting next to each other in the memory). The way a `Vector` will look in the memory is shown in the preceding diagram.

Next, we use our `Vector` object definition to create a Vector instance called `v`:

```
Vector v; // Create a Vector instance called v
```

Once you have an instance of a `Vector`, you can access the variables inside it using what we call **dot syntax**. You access the variable `x` on Vector `v` using `v.x`. The `struct` Vector definition doesn't actually create a Vector object, it just defines the object type. You can't do `Vector.x = 1`. Which object instance are you talking about? the C++ compiler will ask. You need to create a Vector instance first, such as Vector `v`. That creates an instance of a Vector and names it `v`. Then, you can do assignments on the `v` instance, such as `v.x = 0`.

We then use this instance to write values into `v`:

```
v.x=20, v.y=30, v.z=40; // assign some values
```

> TIP: We used commas in the preceding code to initialize a bunch of variables on the same line. This is okay in C++. Although you can do each variable on its own line, the approach shown here is okay too.

This makes `v` look as in the preceding image. Then, we print them out:

```
cout << "A 3-space vector at " << v.x << ", " << v.y << ", " <<
   v.z << endl;
```

In both the lines of code here, we access the individual data members inside the object by simply using a dot (`.`); `v.x` refers to the `x` member inside the object `v`. Each Vector object will have exactly three floats inside it: one called `x`, one called `y`, and one called `z`.

Exercise – player

Define a C++ data struct for a `Player` object. Then, create an instance of your `Player` struct and fill each of the data members with values.

Solution

Let's declare our `Player` object. We want to group together everything to do with the player into the `Player` object. We do this so that the code is neat and tidy. The code you read in Unreal Engine will use objects such as these everywhere, so pay attention:

```
struct Player
{
  string name;
  int hp;
  Vector position;
}; // Don't forget this semicolon at the end!
int main()
{
  // create an object of type Player,
  Player me; // instance named 'me'
  me.name = "William";
  me.hp = 100;
  me.position.x = me.position.y = me.position.z=0;
}
```

TIP

The line `me.position.x = me.position.y = me.position.z=0;` means `me.position.z` is set to `0`, and then that value is passed on to set `me.position.y` to 0, and then it is passed along and sets `me.position.x` to `0`.

The `struct Player` definition is what tells the computer how a `Player` object is laid out in memory.

TIP

I hope you noticed the mandatory semicolon at the end of the struct declaration. Struct object declarations need to have a semicolon at the end, but functions do not (we'll go over functions later). This is just a C++ rule that one must remember.

Inside a `Player` object, we declared a string for the player's name, a float for their hp, and a `Vector` object for their complete `x`, `y`, and `z` position.

When I say object, I mean a C++ struct (later, we will introduce the term *class*).

Wait! We put a Vector object inside a Player object! Yes, you can do that. Just make sure the Vector is defined in the same file.

After the definition of what a `Player` object has inside it, we actually create a `Player` object instance called `me` and assign it some values.

Pointers

A particularly tricky concept to grasp is the concept of pointers. Pointers aren't that hard to understand but can take a while to get a firm handle on. Pointers basically contain an address in memory where an object is stored, so they "point to" the object in memory.

Say we have, as before, declared a variable of the type `Player` in memory:

```
Player me;
me.name = "William";
me.hp = 100;
```

We now declare a pointer to the `Player`:

```
Player* ptrMe;               // Declaring a pointer to
                             // a Player object
```

The `*` changes the meaning of the variable type. The `*` is what makes `ptrMe` a pointer to a `Player` object instead of a regular `Player` object.

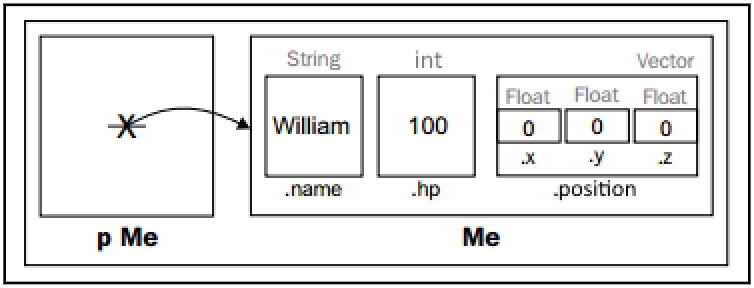
We now want to link `ptrMe` to `me`:

```
ptrMe = &me;                  // LINKAGE
```

TIP

This linkage step is very important. If you don't link the pointer to an object before you use the pointer, you will get a memory access violation—an error that you are trying to access memory that you didn't set, so it could contain random data or even part of another program!

The `ptrMe` pointer now points to the same object as `me`. Changing the values of the variables in the object `ptrMe` points to will change them in `me`, as shown in the following diagram:



What can pointers do?

When we set up the linkage between the pointer variable and what it is pointing to, we can manipulate the variable that it is pointed to through the pointer.

One use of pointers is to refer to the same object from several different locations in the code. You may want to store a pointer to it locally if you'll be constantly trying to access it, to make it easier to access. The `Player` object is a good candidate for being pointed to, since many places in your code could be accessing it constantly.

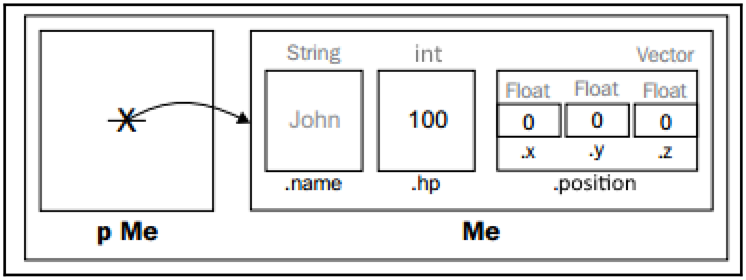
You can create as many pointers as you wish to the same object, but you'll want to keep track of them all (unless you use smart pointers, which we will get to later). Objects that are pointed to do not necessarily know that they are being pointed at, but changes can be made to the object through the pointers.

For instance, say the player got attacked. A reduction in their hp will be the result, and this reduction will be done using the pointer, as shown in the following code:

```
ptrMe->hp -= 33;      // reduced the player's hp by 33
ptrMe->name = "John";// changed his name to John
```

When using a pointer, you need to use `->` instead of `.` to access the variables in the object pointed to.

Here's how the `Player` object looks now:



So, we changed `me.name` by changing `ptrMe->name`. Because `ptrMe` points to `me`, changes through `ptrMe` affect `me` directly.

Address of operator (&)

Notice the use of the `&` symbol in the preceding code example. The `&` operator gets the memory address where a variable is stored. A variable's memory address is a location in the computer memory space that is reserved to store the variable's value. C++ is able to get the memory address of any object in your program's memory. The address of a variable is unique and also kind of random.

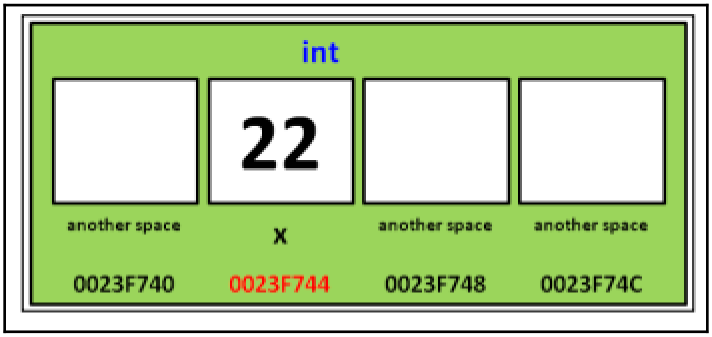
Say we print the address of an integer variable, `x`, as follows:

```
int x = 22;
cout << &x << endl; // print the address of x
```

On the first run of the program, my computer prints the following:

```
0023F744
```

This number (the value of `&x`) is just the memory cell where the `x` variable is stored. What this means is that in this particular launch of the program, the `x` variable is located at memory cell number `0023F744`, as shown in the following diagram:



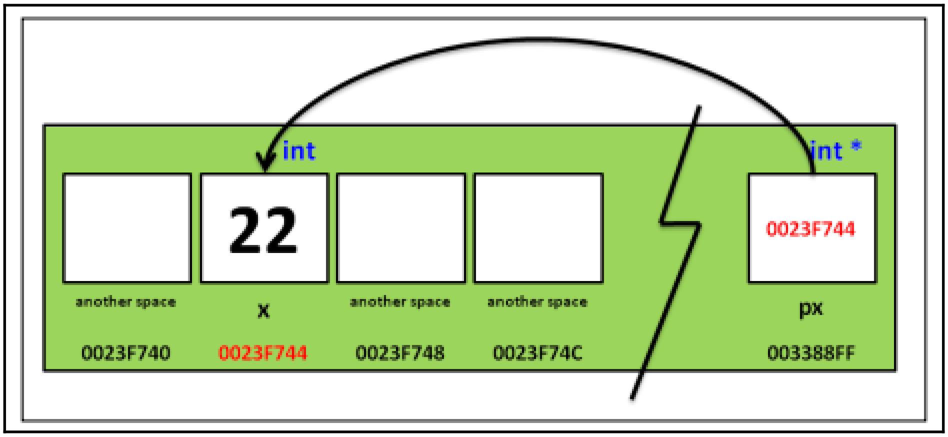
TIP

You may wonder why the preceding number contains an `F`. Addresses are in hexadecimal (base 16) so since you run out of numerical digits after 9, but you can't really fit two digits in 1, you set the values that would be 10-15 to A-F instead. So A = 10, B = 11, and in this case F = 15.

Now, create and assign a pointer variable to the address of `x`:

```
int *px;
px = &x;
```

What we're doing here is storing the memory address of `x` inside the `px` variable. So, we are metaphorically pointing to the `x` variable using another different variable called `px`. This might look similar to what is shown in the following diagram:



Here, the `px` variable has the address of the `x` variable inside it. In other words, the `px` variable is a reference to another variable. De-referencing `px` means to access the variable that `px` is referencing. De-referencing is done using the `*` symbol:

```
cout << *px << endl;
```

Using nullptr

The `nullptr` variable is a pointer variable with the value `0`. In general, most programmers like to initialize pointers to `nullptr` (`0`) on the creation of new pointer variables. Computer programs, in general, can't access the memory address `0` (it is reserved), so if you try to reference a null pointer, your program will crash.

TIP

Pointer Fun with Binky is a fun video about pointers. Take a look at `http://www.youtube.com/watch?v=i49_SNt4yfk`.

Smart pointers

Pointers can be hard to manage. Once we start creating and deleting new objects later in this book, we may not know where all the pointers are that point to a specific object. It can be too easy to delete an object that another pointer is still using (leading to a crash), or to stop pointing to an object from the only pointer to it and leave it floating in memory with nothing referencing it (this is called a memory leak, and will slow down your computer).

Smart pointers keep track of how many references exist to a specific object, and will automatically increment or decrement this number as things change in the code. This makes it much easier to control what is happening, and in real-world programming it is preferable to using regular pointers when possible.

People used to have to write their own smart pointers, but not since C++ 11. There is now a `shared_ptr` template available (we'll talk about templates and the STL later). This will automatically keep track of pointers to an object, and will automatically delete that object if nothing else is referencing it, preventing memory leaks. This is why it is better to use smart pointers than pointers, since regular pointers could wind up pointing to objects that have been deleted elsewhere in the code.

Input and output

In programming, you constantly have to pass information to the user, or get information from the user. For simple cases, such as the ones we will be starting with (and many times for finding errors later), you need to input and output standard text and numbers. C++ makes this easy.

The cin and cout objects

We've already seen how `cout` works in previous examples. The `cin` object is the way C++ traditionally takes input from the user into the program. The `cin` object is easy to use, because it looks at the type of variable it will put the value into and uses that to determine the type being put inside it. For example, say we want to ask the user his age and store it in an `int` variable. We can do that as follows:

```
cout << "What is your age?" << endl;
int age;
cin >> age;
```

When you run this, it will print `What is your age?` and wait for your response. Type in a response and hit *Enter* to input it. You might want to try typing in other things besides `int` variables just to see what happens!

The printf() function

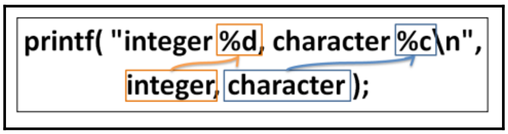
Although we have used `cout` to print out variables so far, you should also know about another common function that is used to print to the console. This function is called the `printf` function, and it originally comes from C. The `printf` function is included in the `<iostream>` library, so you don't have to `#include` anything extra to use it. Some people in the gaming industry prefer `printf` to `cout`, so let's introduce it.

Let's proceed to how `printf()` works, as shown in the following code:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  char character = 'A';
  int integer = 1;
  printf( "integer %d, character %c\n", integer, character );
}
```

> **Downloading the example code**
>
> TIP
>
> You can download the example code files from your account at `http://www.packtpub.com` for all the Packt books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

We start with a format string. The format string is like a picture frame, and the variables will get plugged in at the locations of the `%` in the format string. Then, the entire thing gets dumped out to the console. In the preceding example, the integer variable will be plugged into the location of the first `%` (`%d`), and the character will be plugged into the location of the second `%` (`%c`), as shown in the following screenshot:



You have to use the right format code to get the output to format correctly; take a look at the following table:

Data type	Format code
`Int`	`%d`
`Char`	`%c`
`String`	`%s`

To print a C++ string, you must use the `string.c_str()` function:

```
string s = "Hello"; printf( "string %s\n", s.c_str() );
```

The `s.c_str()` function accesses the C pointer to the string, which `printf` needs.

If you use the wrong format code, the output won't appear correctly or the program might crash.

You may also find cases where you need to use this type of formatting to set up strings, so it is good to know. But if you'd prefer to avoid having to remember these different format codes, just use `cout`. It will figure the type out for you. Just make sure you use whatever standard the company you eventually work for prefers. It's generally a good idea to do that with most things in programming.

Exercise

Ask the user their name and age and take them in using `cin`. Then, issue a greeting for them at the console using `printf()` (not `cout`).

Solution

This is how the program will look:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  cout << "Name?" << endl;
  string name;
  cin >> name;
  cout << "Age?" << endl;
  int age;
  cin >> age;
  //Change to printf:
  cout << "Hello " << name << " I see you have attained " << age
   << " years. Congratulations." << endl;
}
```

TIP

A string is actually an object type. Inside, it is just a bunch of chars!

Namespaces

We've seen namespaces so far in the case of `std`, and we've mostly avoided that issue by putting the following at the top of the files:

```
using namespace std;
```

But, you should know what that means for the future.

Namespaces are ways to group together code that is related, and it allows you to use the same variable names in different namespaces without any naming conflicts (unless of course you put `using namespace` for both at the top, which is why many people prefer to not use that).

You can create your own namespace in a C++ file like this:

```
namespace physics {
    float gravity = 9.80665;
    //Add the rest of your your physics related code here...
}
```

Once you've created your namespace, you can then access that code like this:

```
float g = physics::gravity;
```

Or, you can put in a using statement at the top (just make sure the name isn't used for something else). But, in general you don't want to use this for more complex programs, since a namespace allows you to reuse the same variable name in different namespaces, so if you use it with a namespace with a variable in it that has the same name as one in the current namespace and try to access it, the compiler won't know which one you are referring to, which would cause a conflict.

Summary

In this chapter, we spoke about variables and memory. We talked about mathematical operations on variables and how simple they were in C++.

We also discussed how arbitrarily complex data types can be built using a combination of these simpler data types, such as floats, integers, and characters. Constructions such as this are called objects. In the next chapter, we will start talking about what we can do with these objects!

3

If, Else, and Switch

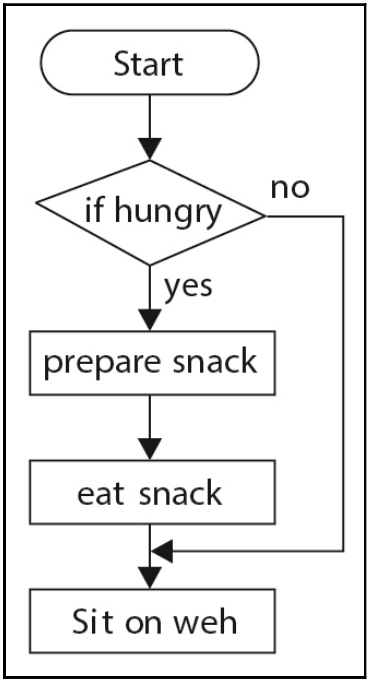
In the previous chapter, we discussed the importance of memory and how it can be used to store data inside a computer. We spoke about how memory is reserved for your program using variables, and how we can include different types of information in our variables.

In this chapter, we will talk about how to control the flow of our program and how we can change what code gets executed by branching the code using control flow statements. Here, we'll discuss the different types of control flow, as follows:

- `If` statements
- How to check whether things are equal using the == operator
- `else` statements
- How to test for inequalities (that is, how to check whether one number is greater or smaller than another using the >, >=, <, <=, and `!=` operators)
- Using logical operators (such as not (`!`), and (`&&`), or (`||`))
- Branching in more than two ways:
    - The `else if` statement
    - The `switch` statement
- Our first example project with Unreal Engine

Branching

The computer code we wrote in `Chapter 2`, *Variables and Memory*, went in one direction: straight down. Sometimes, we might want to be able to skip parts of the code. We might want the code to be able to branch in more than one direction. Schematically, we can represent this in the following manner:



In other words, we want the option to not run certain lines of code under certain conditions. The preceding diagram is called a flowchart. According to this flowchart, if, and only if, we are hungry then we will go prepare a sandwich, eat it, and then go and rest on the couch. If we are not hungry, then there is no need to make a sandwich, so we will simply rest on the couch.

We'll only use flowcharts in this book sometimes, but in UE4, you can even use flowcharts to program your game (using something called blueprints).

This book is about C++ code, so we will always transform our flowcharts into actual C++ code in this book.

Controlling the flow of your program

Ultimately, what we want is the code to branch in one way under certain conditions. Code commands that change which line of code gets executed next are called control flow statements. The most basic control flow statement is the `if` statement. To be able to code `if` statements, we first need a way to check the value of a variable.

So, to start, let's introduce the == symbol, which is used to check the value of a variable.

The == operator

In order to check whether two things are equal in C++, we need to use not one but two equal signs (==) one after the other, as shown here:

```
int x = 5; // as you know, we use one equals sign
int y = 4; // for assignment..
// but we need to use two equals signs
// to check if variables are equal to each other
cout << "Is x equal to y? C++ says: " << (x == y) << endl;
```

If you run the preceding code, you will notice that the output is the following:

```
Is x equal to y? C++ says: 0
```

In C++, `1` means true and `0` means false. If you want the words `true` or `false` to appear instead of `1` and `0`, you can use the `boolalpha` stream manipulator in the `cout` line of code, as shown here:

```
cout << "Is x equal to y? C++ says: " << boolalpha <<
        (x == y) << endl;
```

The == operator is a type of comparison operator. The reason why C++ uses == to check for equality and not just = is that we already used up the = symbol for the assignment operator! (see the *More on variables* section in `Chapter 2`, *Variables and Memory*). If we use a single = sign, C++ will assume that we want to overwrite `x` with `y`, not compare them.

Coding if statements

Now that we have the double equals sign under our belt, let's code the flowchart. The code for the preceding flowchart diagram is as follows:

```
bool isHungry = true;  // can set this to false if not
                       // hungry!
if( isHungry == true ) // only go inside { when isHungry is true
{
  cout << "Preparing snack.." << endl;
  cout << "Eating .. " << endl;
}
cout << "Sitting on the couch.." << endl;
```

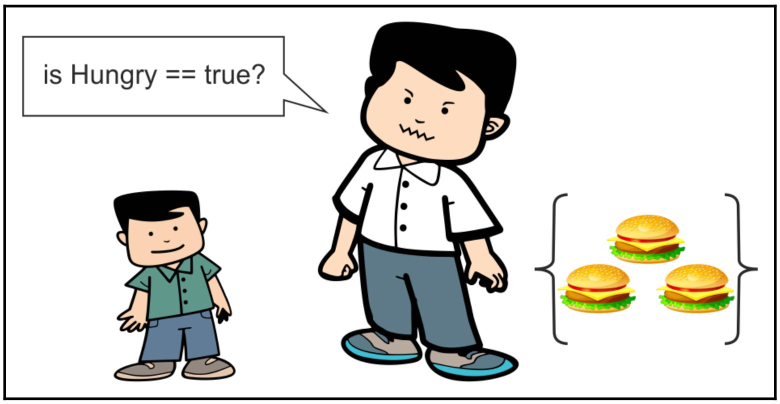
> This is the first time we are using a `bool` variable! A `bool` variable either holds the value `true` or the value `false`.

First, we start with a `bool` variable called `isHungry` and just set it to `true`.

Then, we use an `if` statement, as follows:

```
if( isHungry == true )
```

The `if` statement acts like a guard on the block of code below it (remember that a block of code is a group of code encased within `{` and `}`):



You can only read the code between `{` and `}` if `isHungry==true`.

You can only get at the code inside the curly braces when `isHungry == true`. Otherwise, you will be denied access and forced to skip over that entire block of code.

> Basically, anything that can be evaluated as a boolean can go inside `if (boolean)`. So, we can achieve the same effect by simply writing the following line of code:
> `if( isHungry ) // only go here if isHungry is true`
> This can be used as an alternative for the following:
> `if( isHungry == true )`

The reason people might use the `if( isHungry )` form is to avoid the possibility of making mistakes. Writing `if( isHungry = true )` by accident will set `isHungry` to true every time the `if` statement is hit! To avoid this possibility, we can just write `if( isHungry )` instead. Alternatively, some (wise) people use what are called Yoda conditions to check an `if` statement: `if( true == isHungry )`. The reason we write the `if` statement in this way is that, if we accidentally write `if( true = isHungry )`, this will generate a compiler error, catching the mistake.

Try running this code segment to see what I mean:

```
int x = 4, y = 5;
cout << "Is x equal to y? C++ says: " << (x = y) << endl; //bad!
// above line overwrote value in x with what was in y,
// since the above line contains the assignment x = y
// we should have used (x == y) instead.
cout << "x = " << x << ", y = " << y << endl;
```

The following lines show the output of the preceding lines of code:

```
Is x equal to y? C++ says: 5
x = 5, y = 5
```

The line of code that has `(x = y)` overwrites the previous value of `x` (which was 4) with the value of `y` (which is 5). Although we were trying to check whether `x` equals `y`, what happened in the previous statement was that `x` was assigned the value of `y`.

Coding else statements

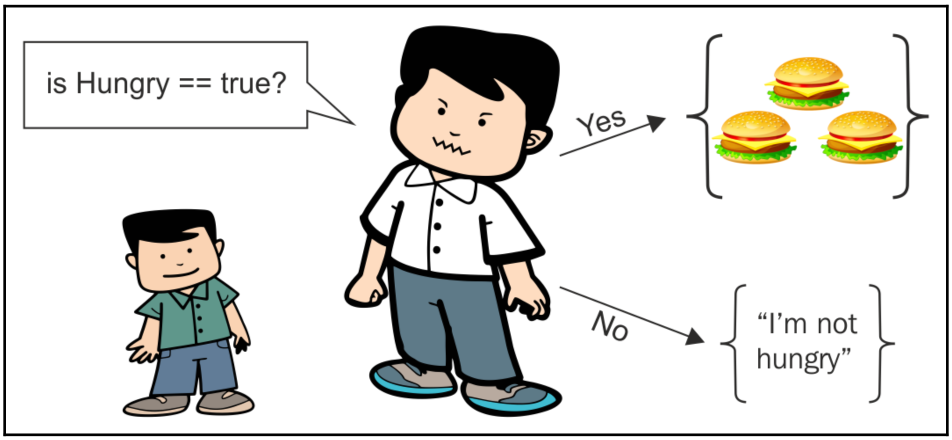
The `else` statement is used to have our code do something in the case that the `if` portion of the code does not run.

For example, say we have something else that we'd like to do in case we are not hungry, as shown in the following code snippet:

```
bool isHungry = true;
if( isHungry )       // notice == true is implied!
{
  cout << "Preparing snack.." << endl;
  cout << "Eating .. " << endl;
}
else                 // we go here if isHungry is FALSE
{
  cout << "I'm not hungry" << endl;
}
cout << "Sitting on the couch.." << endl;
```

There are a few important things that you need to remember about the `else` keyword, as follows:

- An `else` statement must always immediately follow after an `if` statement. You can't have any extra lines of code between the end of the `if` block and the corresponding `else` block.
- A program can never execute both the `if` and the corresponding `else` block. It's always one or the other:



The else statement is the way you will go if `isHungry` is not equal to true.

You can think of the `if/else` statements as a guard diverting people to either the left or the right. Each person will either go toward the food (when `isHungry==true`), or they will go away from the food (when `isHungry==false`).

Testing for inequalities using other comparison operators (>, >=, <, <=, and !=)

Other logical comparisons can be easily done in C++. The > and < symbols mean just what they do in math. They are the greater than (>) and less than (<) symbols. >= has the same meaning as the ≥ symbol in math. <= is the C++ code for ≤. Since there isn't a ≤ symbol on the keyboard, we have to write it using two characters in C++. `!=` is how we say *"not equal to"* in C++. So, for example, say we have the following lines of code:

```
int x = 9;
int y = 7;
```

We can ask the computer whether `x > y` or `x < y`, as shown here:

```
cout << "Is x greater than y? " << (x > y) << endl;
cout << "Is x greater than OR EQUAL to y? " << (x >= y) << endl;
cout << "Is x less than y? " << (x < y) << endl;
cout << "Is x less than OR EQUAL to y? " << (x <= y) << endl;
cout << "Is x not equal to y? " << (x != y) << endl;
```

TIP

We need the brackets around the comparisons of `x` and `y` because of something known as operator precedence. If we don't have the brackets, C++ will get confused between the << and < operators. It's weird and you will better understand this later, but you need C++ to evaluate the `(x < y)` comparison before you output the result (<<). There is an excellent table available for reference at `http://en.cppreference.com/w/cpp/language/operator_precedence`.

Using logical operators

Logical operators allow you to do more complex checks, rather than checking for a simple equality or inequality. Say, for example, the condition to gain entry into a special room requires the player to have both the red and green keycards. We want to check whether two conditions hold true at the same time. To do this type of complex logic statement check there are three additional constructs that we need to learn: the not (`!`), and (`&&`), and or (`||`) operators.

The not (!) operator

The `!` operator is handy to reverse the value of a `boolean` variable. Take the example of the following code:

```
bool wearingSocks = true;
if( !wearingSocks ) // same as if( false == wearingSocks )
{
         cout << "Get some socks on!" << endl;
 }
else
{
        cout << "You already have socks" << endl;
}
```

The `if` statement here checks whether or not you are wearing socks. Then, you are issued a command to get some socks on. The `!` operator reverses the value of whatever is in the `boolean` variable to be the opposite value.

We use something called a truth table to show all the possible results of using the `!` operator on a `boolean` variable, as follows:

`wearingSocks`	`!wearingSocks`
`true`	`false`
`false`	`true`

So, when `wearingSocks` has the value `true`, `!wearingSocks` has the value `false` and vice versa.

Exercises

1. What do you think will be the value of `!!wearingSocks` when the value of `wearingSocks` is `true`?
2. What is the value of `isVisible` after the following code is run?

```
bool hidden = true;
bool isVisible = !hidden;
```

Solutions

1. If `wearingSocks` is `true`, then `!wearingSocks` is `false`. Therefore, `!!wearingSocks` becomes `true` again. It's like saying "I am not not hungry." Not not is a double negative, so this sentence means that I am actually hungry.
2. The answer to the second question is `false`. The value of `hidden` was `true`, so `!hidden` is `false`. The `false` value then gets saved into the `isVisible` variable. But the value of `hidden` itself remains `true`.

> TIP
>
> The `!` operator is sometimes colloquially known as a bang. The preceding bang-bang operation (`!!`) is a double negative and a double logical inversion. If you bang-bang a `bool` variable, there is no net change to the variable.
>
> Of course, you can use these on an `int` and in that case, if the `int` is set to zero, `! int` will be `true`, and if it is greater than zero, `! int` will be `false`. Therefore, if you bang-bang that `int` variable, and the `int` value is greater than zero, it is reduced to a simple `true`. If the `int` value is 0 already, it is reduced to a simple `false`.

The and (&&) operator

Say we only want to run a section of the code if two conditions are `true`. For example, we are only dressed if we are wearing both socks and clothes. You can use the following code to check this:

```
bool wearingSocks = true;
bool wearingClothes = false;
if( wearingSocks && wearingClothes )// && requires BOTH to be true
{
        cout << "You are dressed!" << endl;
}
else
{
        cout << "You are not dressed yet" << endl;
}
```

The or (||) operator

We sometimes want to run a section of the code if either one of the variables is `true`.

So, for example, say the player wins a certain bonus if they find either a special star in the level or the time that they take to complete the level is less than 60 seconds. In this case, you can use the following code:

```
bool foundStar = false;
float levelCompleteTime = 25.f;
float maxTimeForBonus = 60.f;
// || requires EITHER to be true to get in the { below
if( foundStar || (levelCompleteTime < maxTimeForBonus) )
{
        cout << "Bonus awarded!" << endl;
}
else
{
        cout << "No bonus." << endl;
}
```

TIP

You may notice that I added parentheses around `levelCompleteTime < maxTimeForBonus`. While precedence rules may let you add longer statements without them, I've found it can be better to just add them if you have any doubt. It's better safe than sorry (and may be a little clearer to someone else looking at it later).

Exercise

By now, you should have noticed that the best way to get better at programming is by doing it. You have to practice programming a lot to get significantly better at it.

Create two integer variables, called `x` and `y`, and read them in from the user. Write an `if`/`else` statement pair that prints the name of the bigger-valued variable.

Solution

The solution to the preceding exercise is shown in the following block of code:

```
int x, y;
cout << "Enter two numbers (integers), separated by a space " << endl;
cin >> x >> y;
if( x < y )
```

```
{
  cout << "x is less than y" << endl;
}
else
{
  cout << "x is greater than y" << endl;
}
```

Don't type a letter when `cin` expects a number. If that happens, `cin` can fail and give a bad value to your variable.

Branching code in more than two ways

In the previous sections, we were only able to make the code branch in one of the two ways. In pseudocode, we had the following code:

```
if( some condition is true )
{
  execute this;
}
else // otherwise
{
  execute that;
}
```

Pseudocode is *fake code*. Writing pseudocode is a great way to brainstorm and plan out your code, especially if you are not quite used to C++.

This code is a little bit like a metaphorical fork in the road, with only one of two directions to choose from.

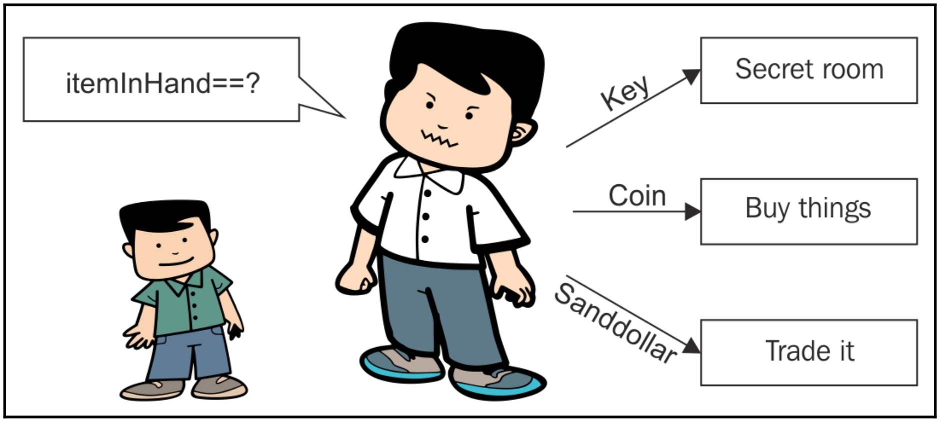
Sometimes, we might want to branch the code in more than just two directions. We might want the code to branch in three ways, or even more. For example, say the direction in which the code goes depends on what item the player is currently holding. The player can be holding one of three different items: a coin, key, or sand dollar. And C++ allows that! In fact, in C++, you can branch in any number of directions that you wish.

The else if statement

The `else if` statement is a way to code in more than just two possible branch directions. In the following code example, the code will go in one of the three different ways, depending on whether the player is holding the `Coin`, `Key`, or `Sanddollar` object:

```
#include <iostream>
using namespace std;
int main()
{
  enum Item  // This is how enums come in handy!
  {
    Coin, Key, Sanddollar // variables of type Item can have
    // any one of these 3 values
  };
  Item itemInHand = Key;  // Try changing this value to Coin,
                          // Sanddollar
  if( itemInHand == Key )
  {
    cout << "The key has a lionshead on the handle." << endl;
    cout << "You got into a secret room using the Key!" << endl;
  }
  else if( itemInHand == Coin )
  {
    cout << "The coin is a rusted brassy color. It has a picture
     of a lady with a skirt." << endl;
    cout << "Using this coin you could buy a few things" << endl;
  }
  else if( itemInHand == Sanddollar )
  {
    cout << "The sanddollar has a little star on it." << endl;
    cout << "You might be able to trade it for something." <<
     endl;
  }
  return 0;
}
```

Note that the preceding code only goes in one of the three separate ways! In an `if`, `else`, and `else if` series of checks, we will only ever go into one of the blocks of code.



Exercise

Use a C++ program to answer the questions that follow the code. Be sure to try these exercises in order to gain fluency with these equality operators:

```
#include <iostream>
using namespace std;
int main()
{
  int x;
  int y;
  cout << "Enter an integer value for x:" << endl;
  cin >> x; // This will read in a value from the console
  // The read in value will be stored in the integer
  // variable x, so the typed value better be an integer!
  cout << "Enter an integer value for y:" << endl;
  cin >> y;
  cout << "x = " << x << ", y = " << y << endl;
  // *** Write new lines of code here
}
```

Write some new lines of code at the spot that says (`// *** Write new...`):

1. Check whether `x` and `y` are equal. If they are equal, print `x and y are equal`. Otherwise, print `x and y are not equal`.
2. An exercise on inequalities: check whether `x` is greater than `y`. If it is, print `x is greater than y`. Otherwise, print `y is greater than x`.

Solution

To evaluate equality, insert the following code:

```
if( x == y )
{
  cout << "x and y are equal" << endl;
}
else
{
  cout << "x and y are not equal" << endl;
}
```

To check which value is greater, insert the following code:

```
if( x > y )
{
  cout << "x is greater than y" << endl;
}
else if( x < y )
{
  cout << "y is greater than x" << endl;
}
else // in this case neither x > y nor y > x
{
  cout << "x and y are equal" << endl;
}
```

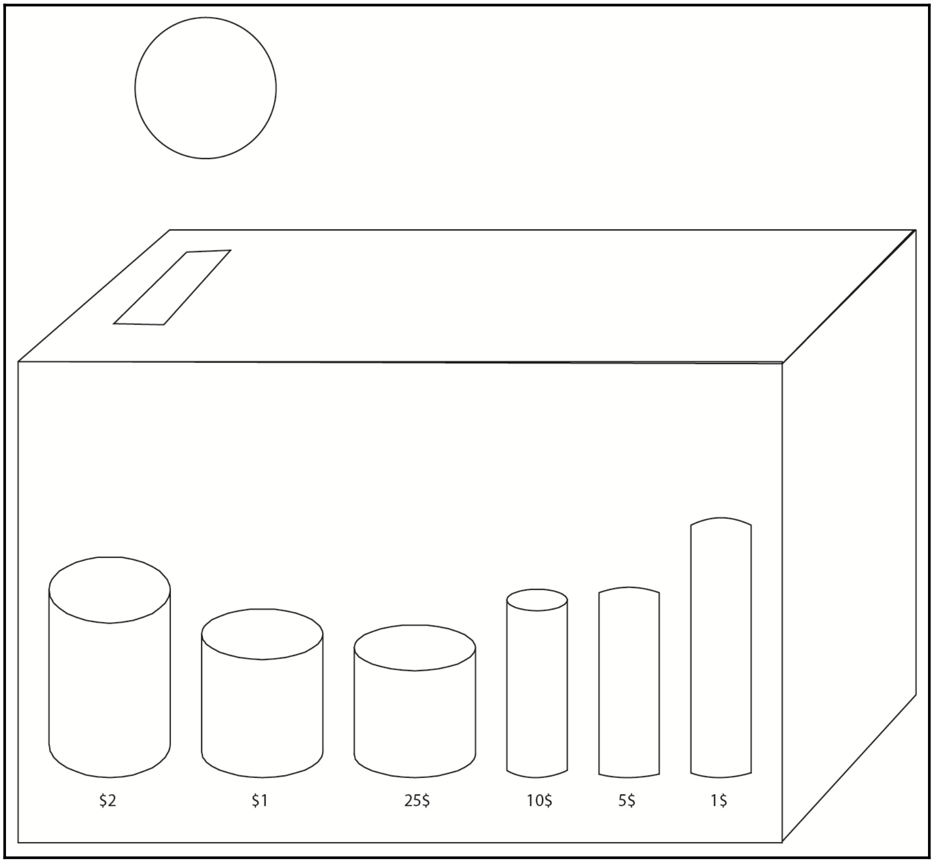
The switch statement

The `switch` statement allows your code to branch in multiple ways. What the `switch` statement will do is look at the value of a variable, and depending on its value, the code will go in a different direction.

We'll also see the `enum` construct here:

```
#include <iostream>
using namespace std;
enum Food  // enums are very useful with switch!
{
  // a variable of type Food can have any of these values
  Fish,
  Bread,
  Apple,
  Orange
};
int main()
{
  Food food = Bread; // Change the food here
  switch( food )
  {
    case Fish:
      cout << "Here fishy fishy fishy" << endl;
      break;
    case Bread:
      cout << "Chomp! Delicious bread!" << endl;
      break;
    case Apple:
      cout << "Mm fruits are good for you" << endl;
      break;
    case Orange:
      cout << "Orange you glad I didn't say banana" << endl;
      break;
    default:  // This is where you go in case none
              // of the cases above caught
      cout << "Invalid food" << endl;
      break;
  }
  return 0;
}
```

Switches are like coin sorters. When you drop a 25-cent coin into a coin sorter, it finds its way into the 25-cent coin pile. Similarly, a `switch` statement will simply allow the code to jump down to the appropriate section. The example of sorting coins is shown in the following diagram:



The code inside the `switch` statement will continue to run (line by line) until the `break;` statement is hit. The `break` statement jumps you out of the `switch` statement. If you leave out the `break;` statements, it will continue to run the code inside the next case statement and won't stop until it either hits a `break;` or the end of the `switch`. If you want to experiment, try taking out all the `break;` statements and see what happens! Take a look at the following diagram to understand how the `switch` works:

```cpp
Food food = Fish; // Change the food here
switch( food )
{
  case Fish:
      cout << "Here fishy fishy fishy" << endl;
      break;
  case Bread:
      cout << "Chomp! Delicious bread!" << endl;
      break;
}
cout << "End of switch" << endl;
```

1. First, the `Food` variable is inspected. What value does it have? In this case, it has `Fish` inside it.
2. The `switch` command jumps down to the correct case label. (If there is no matching case label, `switch` will just be skipped).
3. The `cout` statement is run, and `Here fishy fishy fishy` appears on the console.
4. After inspecting the variable and printing the user response, the `break` statement is hit. This makes us stop running lines of code in the `switch` and exit the `switch`. The next line of code that is run is just what would otherwise have been the next line of code in the program if the `switch` had not been there at all (after the closing curly brace of the `switch` statement). It is the `return 0` that exits the program.

The switch statement versus the if statement

Switches are like the `if` / `else if` / `else` chains from earlier. However, switches can generate code faster than `if` / `else if` / `else if` / `else` chains. Intuitively, switches only jump to the appropriate section of the code to execute. An `if` / `else if` / `else` chain might involve more complicated comparisons (including logical comparisons), which might take more CPU time. The main reason you will use `if` statements is if you are trying to check something that's more complicated than just comparing something in a specific set of values.

An instance of an `enum` is really an `int`. To verify this, print the following code:

```
cout << "Fish=" << Fish <<
   " Bread=" << Bread <<
   " Apple=" << Apple <<
 "Orange=" << Orange << endl;
```

You will see the integer values of the `enum`—just so you know.

Sometimes, programmers want to group multiple values under the same switch `case` label. Say we have an `enum` object as follows:

```
enum Vegetables { Potato, Cabbage, Broccoli, Zucchini };
```

A programmer wants to group all the greens together, so they write a `switch` statement as follows:

```
Vegetable veg = Zucchini;

switch( veg )
{
case Zucchini:             // zucchini falls through because no break
case Broccoli:             // was written here
  cout << "Greens!" << endl;
  break;
default:
  cout << "Not greens!" << endl;
  break;
}
```

In this case, `Zucchini` falls through and executes the same code as `Broccoli`.
The non-green vegetables are in the `default` case label. To prevent a fall-through, you have to remember to insert an explicit `break` statement after each `case` label.

We can write another version of the same switch that does not let Zucchini fall through, by the explicit use of the `break` keyword in the switch:

```
switch( veg )
{
case Zucchini:              // zucchini no longer falls due to break
  cout << "Zucchini is a green" << endl;
  break;// stops case zucchini from falling through
case Broccoli:               // was written here
  cout << "Broccoli is a green" << endl;
  break;
default:
```

```
    cout << "Not greens!" << endl;
    break;
  }
```

Note that it is good programming practice to `break` the `default` case as well, even though it is the last case listed.

Exercise

Complete the following program, which has an `enum` object with a series of mounts to choose from. Write a `switch` statement that prints the following messages for the mount selected:

`Horse`	The steed is valiant and mighty.
`Mare`	This mare is white and beautiful.
`Mule`	You are given a mule to ride. You resent that.
`Sheep`	Baa! The sheep can barely support your weight.
`Chocobo`	Chocobo!

Remember, an `enum` object is really an `int` statement. The first entry in an `enum` object is by default `0`, but you can give the `enum` object any starting value you wish using the `=` operator. Subsequent values in the `enum` object are `ints` arranged in order.

Solution

The solution to the preceding exercise is shown in the following code:

```
#include <iostream>
using namespace std;
enum Mount
{
  Horse=1, Mare, Mule, Sheep, Chocobo
  // Since Horse=1, Mare=2, Mule=3, Sheep=4, and Chocobo=5.
};
int main()
{
  int mount;  // We'll use an int variable for mount
              // so cin works
  cout << "Choose your mount:" << endl;
  cout << Horse << " Horse" << endl;
  cout << Mare << " Mare" << endl;
  cout << Mule << " Mule" << endl;
```

```
  cout << Sheep << " Sheep" << endl;
  cout << Chocobo << " Chocobo" << endl;
  cout << "Enter a number from 1 to 5 to choose a mount" << endl;
  cin >> mount;
    // Describe what happens
    // when you mount each animal in the switch below
  switch( mount )
  {
    default:
      cout << "Invalid mount" << endl;
      break;
  }
return 0;
}
```

Bit-shifted enums

A common thing to do in an `enum` object is to assign a bit-shifted value to each entry:

```
enum   WindowProperties
{
    Bordered    = 1 << 0, // binary 001
    Transparent = 1 << 1, // binary 010
    Modal       = 1 << 2  // binary 100
};
```

The bit-shifted values should be able to combine the window properties. This is how the assignment will look:

```
//   bitwise OR combines properties
WindowProperties   wp = Bordered | Modal;
```

Checking which `WindowProperties` have been set involves a check using `bitwise AND`:

```
//   bitwise AND checks to see if wp is Modal
if( wp   & Modal )
{
    cout << "You are looking at a modal window" << endl;
}
```

TIP

Bit-shifting is a technique that is slightly beyond the scope of this book, but I've included this tip just so you know about it.

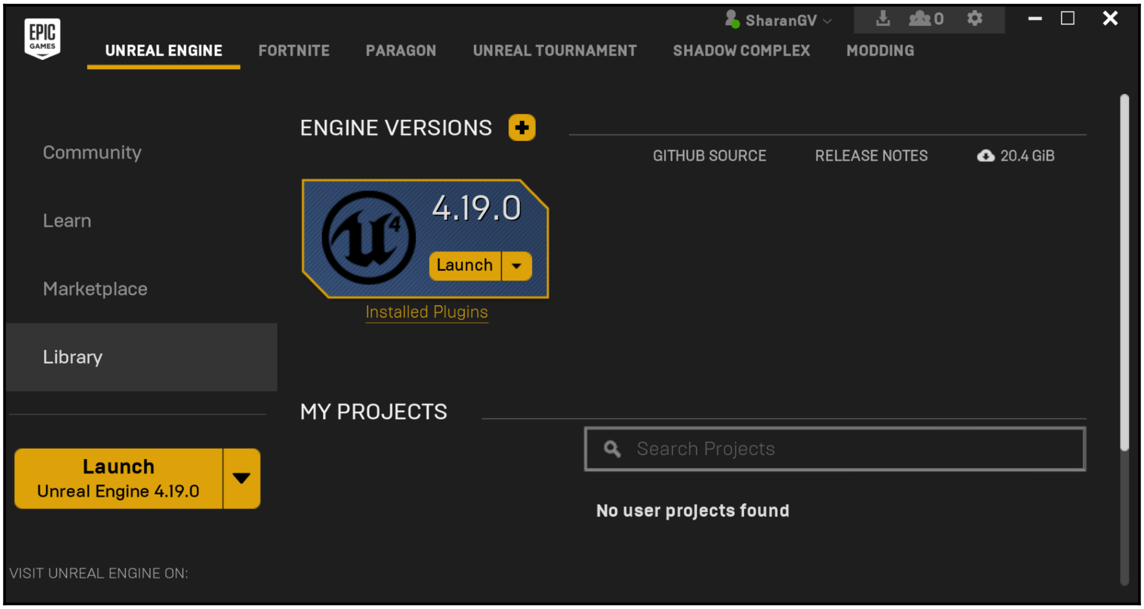
Our first example with Unreal Engine

We need to get started with Unreal Engine.

> A word of warning: when you open your first Unreal project, you will find that the code looks very complicated. Don't get discouraged. Simply focus on the highlighted parts. Throughout your career as a programmer, you will often have to deal with very large code bases containing sections that you do not understand. However, focusing on the parts that you do understand will make this section productive.

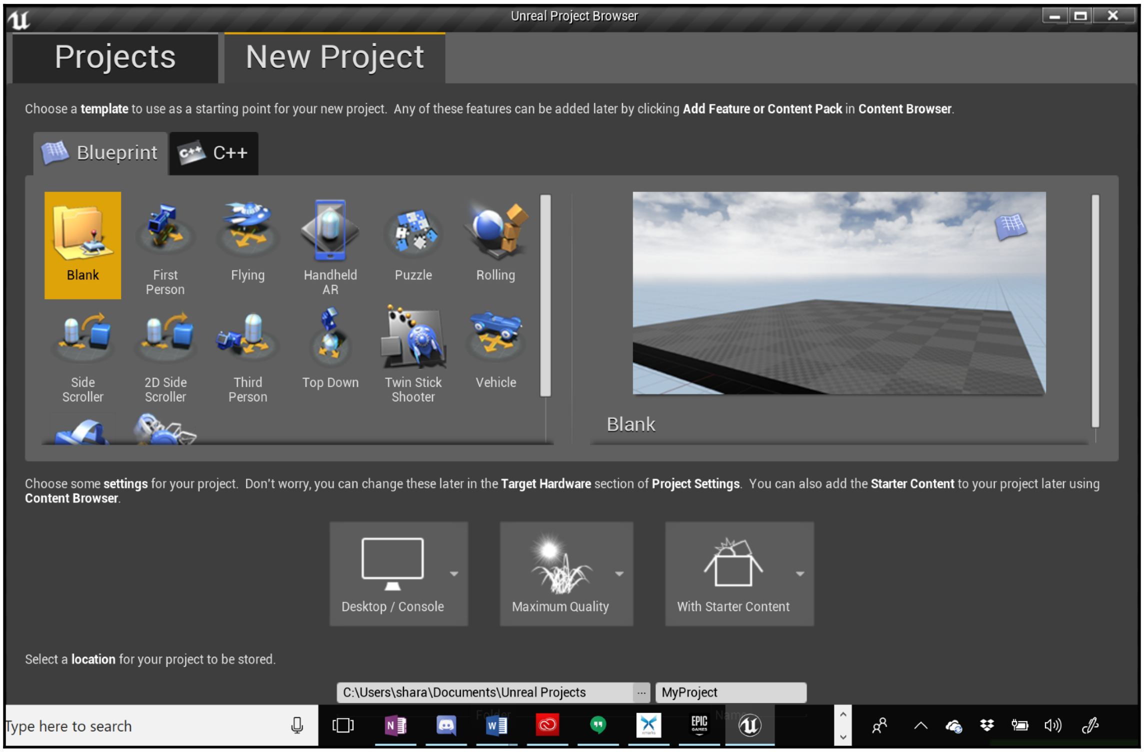
First, you need to download the launcher to install the engine. Go to `https://www.unrealengine.com/en-US/what-is-unreal-engine-4` and when you click **Get Started Now** or **Download**, you will have to create a free account before you can download the launcher.

Once you have downloaded the launcher, open the **Epic Games Launcher** app. Select **Launch Unreal Engine 4.20.X** (there will probably be a new version by the time you read this), as shown in the following screenshot:



> If you don't have the engine installed, you need to go to the **Unreal Engine** tab and download an engine (~7 GB).

Once the engine is launched (which might take a few seconds), you will be in the **Unreal Project Browser** screen, as shown in the following screenshot:



Now, select the **New Project** tab in the UE4 project browser. Choose the **C++** tab and select the **Puzzle** project. This is one of the simpler projects that doesn't have too much code, so it's good to start with. We'll move on to the 3D projects later.

Here are a few things to make a note of in this screen:

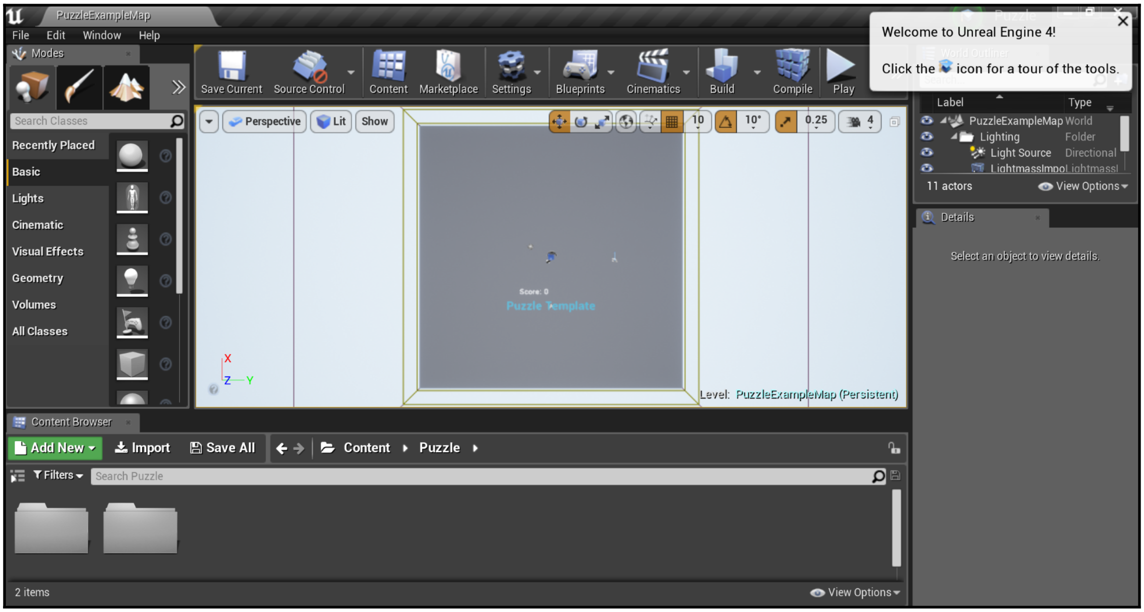
- Be sure you're in the **New Project** tab.
- When you click on **Puzzle**, make sure that it is the one in the **C++** tab, not the **Blueprint** tab.
- Enter a name for your project, `Puzzle`, in the **Name** box (this is important for the example code I will give you to work on later).
- If you want to change the storage folder (such as to a different drive), click the **...** button next to the folder so that the browse window appears. Then, find the directory where you want to store your project.

After you've done all this, select **Create Project**.

TIP

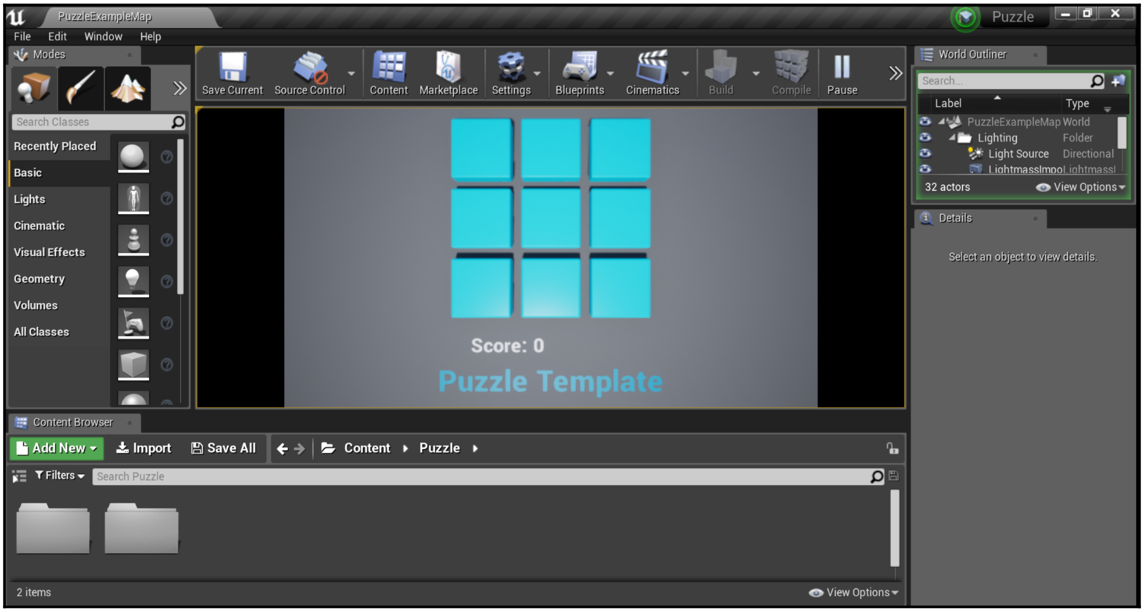
Note: if it tells you it can't create the project because you do not have the Windows 8.1 SDK installed, you can download it from `https://developer.microsoft.com/en-us/windows/downloads/sdk-archive`.

Visual Studio 2017 will open with the code of your project, as well as the Unreal Editor, as shown in the following screenshot:



Looks complicated? Oh boy, it sure is! We'll explore some of the functionality in the toolbars at the side later. For now, just select **Play**, as shown in the preceding screenshot.

This launches the game. This is how it should look:



Now, try clicking on the blocks. As soon as you click on a block, it turns orange, and this increases your score. You can end your play session by clicking **Stop** or hitting *Esc* on your keyboard.

What we're going to do is find the section that does this and change the behavior a little.

Find and open the `PuzzleBlock.cpp` file. Look for **PuzzleBlock** under **C++ Classes** | **Puzzle** and double-click it to open it in your IDE.

TIP: In Visual Studio, the list of files in the project is located inside the **Solution Explorer**. If your **Solution Explorer** is hidden, simply click on **View/Solution Explorer** from the menu at the top.

Inside this file, scroll down to the bottom, where you'll find a section that begins with the following words:

```
void APuzzleBlock::BlockClicked(UPrimitiveComponent* ClickedComp, FKey
ButtonClicked)
```

`APuzzleBlock` is the class name (we'll get into classes later), and `BlockClicked` is the function name. Whenever a puzzle block gets clicked on, the section of code from the starting `{` to the ending `}` is run. Hopefully, exactly how this happens will make more sense later.

It's kind of like an `if` statement in a way. If a puzzle piece is clicked on, then this group of the code is run for that puzzle piece.

We're going to walk through the steps to make the blocks flip colors when they are clicked on (so, a second click will change the color of the block from orange back to blue).

Perform the following steps with the utmost care:

1. Open the `PuzzleBlock.h` file. After the line which has this code:

```
/** Pointer to blue material used on inactive blocks */
  UPROPERTY()
  class UMaterialInstance* BlueMaterial;

  /** Pointer to orange material used on active blocks */
  UPROPERTY()
  class UMaterialInstance* OrangeMaterial;
```

2. Now, open the `PuzzleBlock.cpp` file. Look for the following code:

```
BlueMaterial = ConstructorStatics.BlueMaterial.Get();
OrangeMaterial = ConstructorStatics.OrangeMaterial.Get()
```

3. In `PuzzleBlock.cpp`, replace the contents of the void `APuzzleBlock::BlockClicked` section of code with the following code:

```
void APuzzleBlock::BlockClicked(UPrimitiveComponent* ClickedComp,
FKey ButtonClicked)
{
  // --REPLACE FROM HERE--
  bIsActive = !bIsActive; // flip the value of bIsActive
  // (if it was true, it becomes false, or vice versa)
  if ( bIsActive )
  {
    BlockMesh->SetMaterial(0, OrangeMaterial);
  }
  else
  {
    BlockMesh->SetMaterial(0, BlueMaterial);
  }
  // Tell the Grid
```

```
      if(OwningGrid != NULL)
      {
        OwningGrid->AddScore();
      }
      // --TO HERE--
    }
```

Only replace inside the `void APuzzleBlock::BlockClicked(UPrimitiveComponent* ClickedComp, FKey ButtonClicked)` statement.
Do not replace the line that starts with `void APuzzleBlock::BlockClicked`. You might get an error (if you haven't named your project `Puzzle`). If so, you can start over by creating a new project with the correct name.

Press **Play** to see your changes in action! So, let's analyze this. This is the first line of code:

```
bIsActive = !bIsActive; // flip the value of bIsActive
```

This line of code simply flips the value of `bIsActive`. The `bIsActive` variable is a `bool` variable (it is created in `APuzzleBlock.h`), which keeps track of whether or not the block is active and should be displayed in orange. It's like flipping a switch. If `bIsActive` is `true`, `!bIsActive` will be `false`. So, whenever this line of code is hit (which happens with a click on any block), the `bIsActive` value is reversed (from `true` to `false` or from `false` to `true`).

Let's consider the next block of code:

```
if ( bIsActive )
  {
    BlockMesh->SetMaterial(0, OrangeMaterial);
  }
  else
  {
    BlockMesh->SetMaterial(0, BlueMaterial);
  }
```

We are simply changing the block color. If `bIsActive` is `true`, then the block becomes orange. Otherwise, the block turns blue.

Summary

In this chapter, you learned how to branch the code. Branching makes it possible for the code to go in a different direction instead of going straight down.

In the next chapter, we will move on to a different kind of control flow statement that will allow you to go back and repeat a line of code a certain number of times. The sections of code that repeat will be called loops.

4
Looping

In the previous chapter, we discussed the `if` statement. The `if` statement enables you to put a condition on the execution of a block of code.

In this chapter, we will explore loops, which are code structures that enable you to repeat a block of code under certain conditions. We stop repeating that block of code once the condition becomes false.

In this chapter, we will explore the following topics:

- The while loop
- The do/while loop
- The for loop
- A simple example of a practical loop within Unreal Engine

The while loop

The `while` loop is used to run a section of the code repeatedly. This is useful if you have a set of actions that must be done repeatedly to accomplish some goal. For example, the `while` loop in the following code repeatedly prints the value of the variable `x` as it is incremented from `1` to 5:

```
int x = 1;
while( x <= 5 ) // may only enter the body of the while when x<=5
{
  cout << "x is " << x << endl;
  x++;
}
cout << "Finished" << endl;
```

This is the output of the preceding program:

```
x is 1
x is 2
x is 3
x is 4
x is 5
Finished
```

In the first line of code, an integer variable `x` is created and set to `1`. Then, we go to the `while` condition. The `while` condition says that while `x` is less than or equal to `5`, you must stay in the block of code that follows.

Each iteration of the loop (an iteration means executing everything between `{` and `}` once) gets a little more done from the task (of printing the numbers `1` to `5`). We program the loop to exit automatically once the task is done (when `x <= 5` is no longer true).

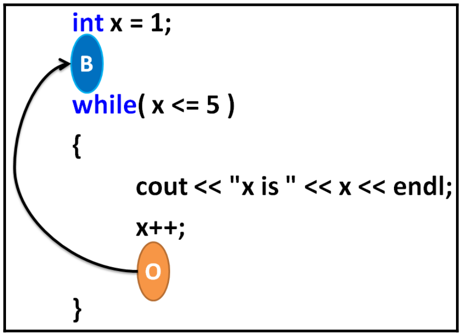
Similar to the `if` statement of the previous chapter, entry into the following block by the `while` loop is only allowed if you meet the condition within the brackets of the `while` loop (in the preceding example, `x <= 5`). You can try mentally subbing an `if` loop in the place of the `while` loop, as shown in the following code:

```
int x = 1;
if( x <= 5 ) // you may only enter the block below when x<=5
{
  cout << "x is " << x << endl;
  x = x + 1;
}
cout << "End of program" << endl;
```

The preceding code sample will only print `x is 1`. So, a `while` loop is exactly like an `if` statement, only it has this special property of automatically repeating itself until the condition between the brackets of the `while` loop becomes false.

I'd like to explain the repetition of the `while` loop using a video game. If you don't know Valve's *Portal*, you should play it, if only to understand loops. Check out `https://www.youtube.com/watch?v=TluRVBhmf8w` for a demo video.

The `while` loops have a kind of magic portal at the bottom, which causes the loop to repeat. The following screenshot illustrates what I mean:



There is a portal at the end of the while loop that takes you back to the beginning

In the preceding screenshot, we loop back from the orange portal (marked `O`) to the blue portal (marked `B`). This is our first time of being able to go back in the code. It is like time travel, only for the code. How exciting!

The only way past a `while` loop block is to not meet the entry condition. In the preceding example, once the value of `x` becomes 6 (so `x <= 5` becomes false), we will not enter the `while` loop again. Since the orange portal is inside the loop, we'll be able to exit the loop once `x` becomes 6.

Infinite loops

You can get stuck inside the same loop forever. Consider the modified program in the following block of code (what do you think will be the output?):

```
int x = 1;
while( x <= 5 ) // may only enter the body of the while when x<=5
{
  cout << "x is " << x << endl;
}
cout << "End of program" << endl;
```

This is how the output will look:

```
x is 1
x is 1
x is 1
.
.
.
(repeats forever)
```

The loop repeats forever because we removed the line of code that changed the value of `x`. If the value of `x` stays the same and is not allowed to increase, we will be stuck inside the body of the `while` loop. This is because the loop's exit condition (the value of `x` becomes 6) cannot be met if `x` does not change inside the loop body.

TIP

Just click the **x** button on the window to close the program.

The following exercises will use all the concepts from the previous chapters, such as the `+=` and decrement operations. If you've forgotten something, go back and reread the previous sections.

Exercises

Let's take a look at a few exercises here:

1. Write a `while` loop that will print the numbers from `1` to `10`
2. Write a `while` loop that will print the numbers from 10 to 1 (backward)
3. Write a `while` loop that will print numbers 2 to 20, incrementing by 2 (for example 2, 4, 6, and 8)
4. Write a `while` loop that will print the numbers 1 to 16 and their squares beside them

The following is an example program output of Exercise 4:

1	1
2	4
3	9
4	16
5	25

Solutions

The code solutions of the preceding exercises are as follows:

1. The solution of the `while` loop that prints the numbers from `1` to `10` is as follows:

```
int x = 1;
while( x <= 10 )
{
  cout << x << endl;
  x++;
}
```

2. The solution of the `while` loop that prints the numbers from `10` to `1` backward is as follows:

```
int x = 10; // start x high
while( x >= 1 ) // go until x becomes 0 or less
{
  cout << x << endl;
  x--; // take x down by 1
}
```

3. The solution of the `while` loop that prints the numbers from `2` to `20` incrementing by `2` is as follows:

```
int x = 2;
while( x <= 20 )
{
  cout << x << endl;
  x+=2; // increase x by 2's
}
```

4. The solution of the `while` loop that prints the numbers from `1` to `16` with their squares is as follows:

```
int x = 1;
while( x <= 16 )
{
  cout << x << "   " << x*x << endl; // print x and it's
   square
  x++;
}
```

The do/while loop

The `do/while` loop is almost identical to the `while` loop. Here's an example of a `do/while` loop that is equivalent to the first `while` loop that we examined:

```
int x = 1;
do
```

```
{
  cout << "x is " << x << endl;
  x++;
} while( x <= 5 ); // may only loop back when x<=5
cout << "End of program" << endl;
```

The only difference here is that we don't have to check the `while` condition on our first entry into the loop. This means that the `do`/`while` loop's body is always executed at least once (where a `while` loop can be skipped entirely if the condition to enter the while `loop` is false when you hit it for the first time).

Here's an example:

```
int val = 5;
while (val < 5)
{
    cout << "This will not print." << endl;
}
do {
    cout << "This will print once." << endl;
} while (val < 5);
```

The for loop

The `for` loop has a slightly different anatomy than the `while` loop, but both are very similar.

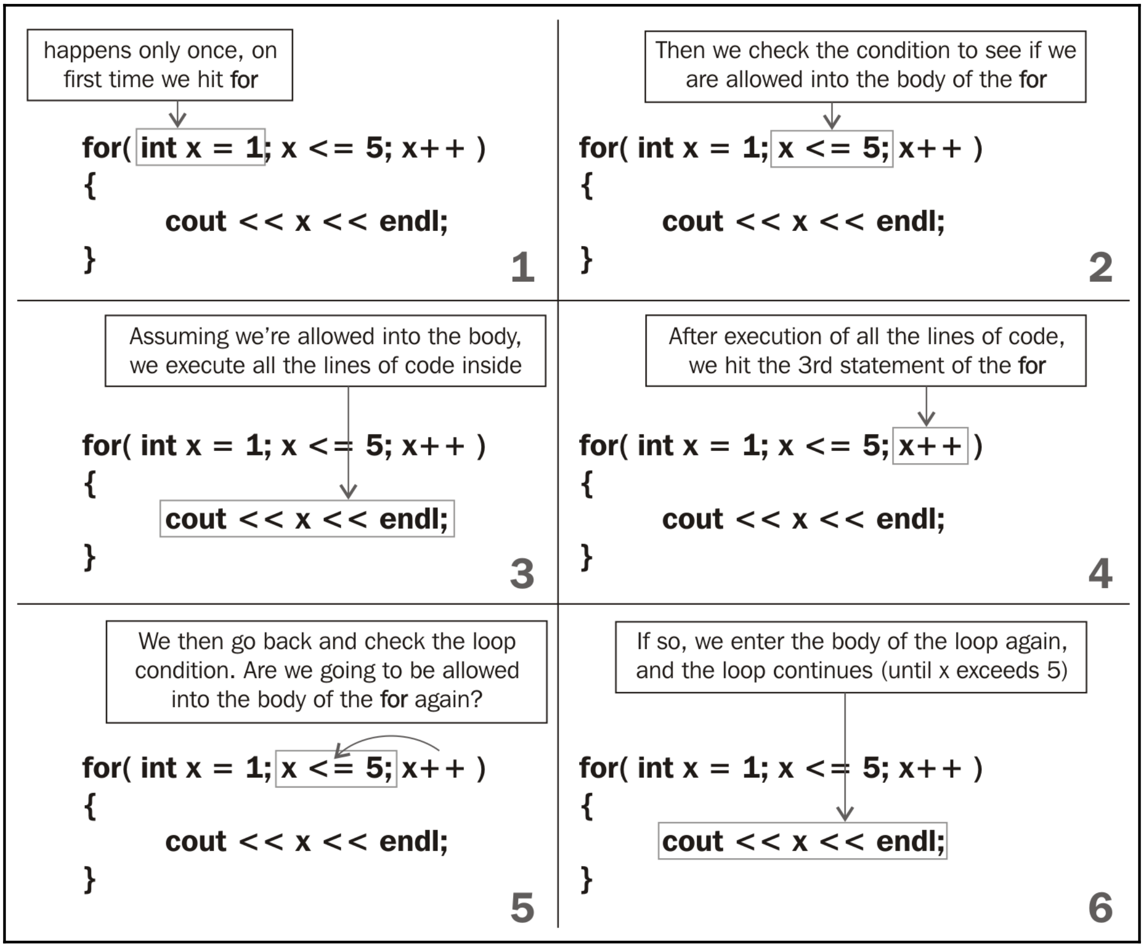
Let's examine the anatomy of a `for` loop as compared to an equivalent `while` loop. Take an example of the following code snippets:

<table>
<tr><th>The <code>for</code> loop</th><th>An equivalent <code>while</code> loop</th></tr>
<tr><td><pre>for(int x = 1; x <= 5; x++)
{
    cout << x << endl;
}</pre></td><td><pre>int x = 1;
while(x <= 5)
{
    cout << x << endl;
    x++;
}</pre></td></tr>
</table>

The `for` loop has three statements inside its parentheses. Let's examine them in order.

The first statement of the `for` loop (`int x = 1;`) only gets executed once, when we first enter the body of the `for` loop. It is typically used to initialize the value of the loop's counter variable (in this case, the variable `x`). The second statement inside the `for` loop (`x <= 5;`) is the loop's repeat condition. As long as `x <= 5`, we must continue to stay inside the body of the `for` loop. The last statement inside the brackets of the `for` loop (`x++;`) gets executed after we complete the body of the `for` loop each time.

The following sequence of diagrams explain the progression of the `for` loop:



Exercises

Let's take a look at some exercises here:

1. Write a `for` loop that will gather the sum of the numbers from `1` to `10`
2. Write a `for` loop that will print the multiples of `6`, from `6` to `30` (6, 12, 18, 24, and 30)
3. Write a `for` loop that will print numbers 2 to 100 in multiples of `2` (for example, 2, 4, 6, 8, and so on)
4. Write a `for` loop that will print numbers `1` to `16` and their squares beside them

Solutions

Here are the solutions for the preceding exercises:

1. The solution for the `for` loop for printing the sum of the numbers from `1` to `10` is as follows:

```
int sum = 0;
for( int x = 1; x <= 10; x++ )
{
  sum += x;
}
cout << sum << endl;
```

2. The solution for the `for` loop for printing multiples of `6` from `6` to `30` is as follows:

```
for( int x = 6; x <= 30; x += 6 )
{
  cout << x << endl;
}
```

3. The solution for the `for` loop for printing numbers from `2` to `100` in multiples of `2` is as follows:

```
for( int x = 2; x <= 100; x += 2 )
{
  cout << x << endl;
}
```

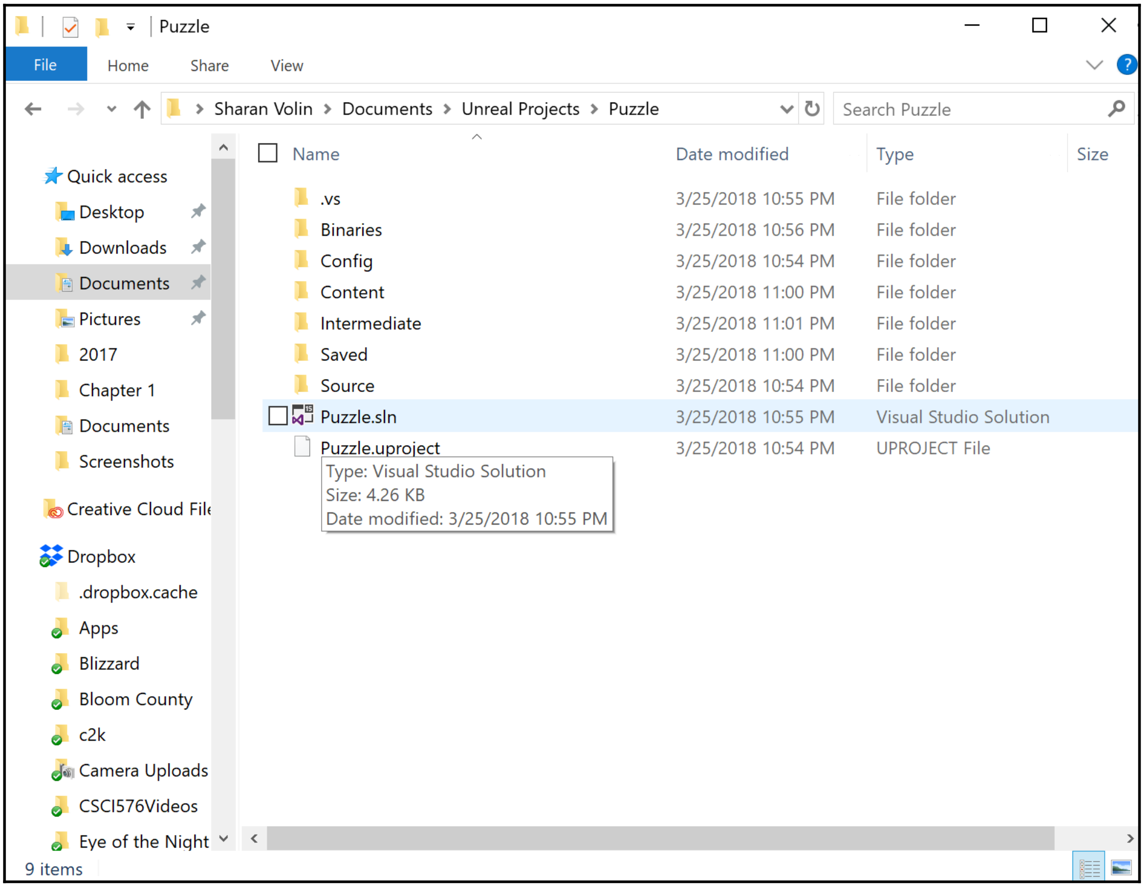
4. The solution for the `for` loop that prints numbers from `1` to `16` and their squares is as follows:

```
for( int x = 1; x <= 16; x++ )
{
  cout << x << " " << x*x << endl;
}
```

Looping with Unreal Engine

In your code editor, open your Unreal `Puzzle` project from `Chapter 3`, *If, Else, and Switch*.

There are several ways to open your Unreal project. On Windows, the simplest way is probably to navigate to the `Unreal Projects` folder (which is present in your user's `Documents` folder by default) and double-click on the `.sln` file in Windows Explorer, as shown in the following screenshot:



In Windows, open the `.sln` file to edit the project code. You can also just open Visual Studio and it will remember which projects you worked on recently and display them so you can click on it from there to open it. You will also need to open the project in the Unreal Editor from the Epic Games Launcher to test it.

Now, open the `PuzzleBlockGrid.cpp` file. Inside this file, scroll down to the section that begins with the following statement:

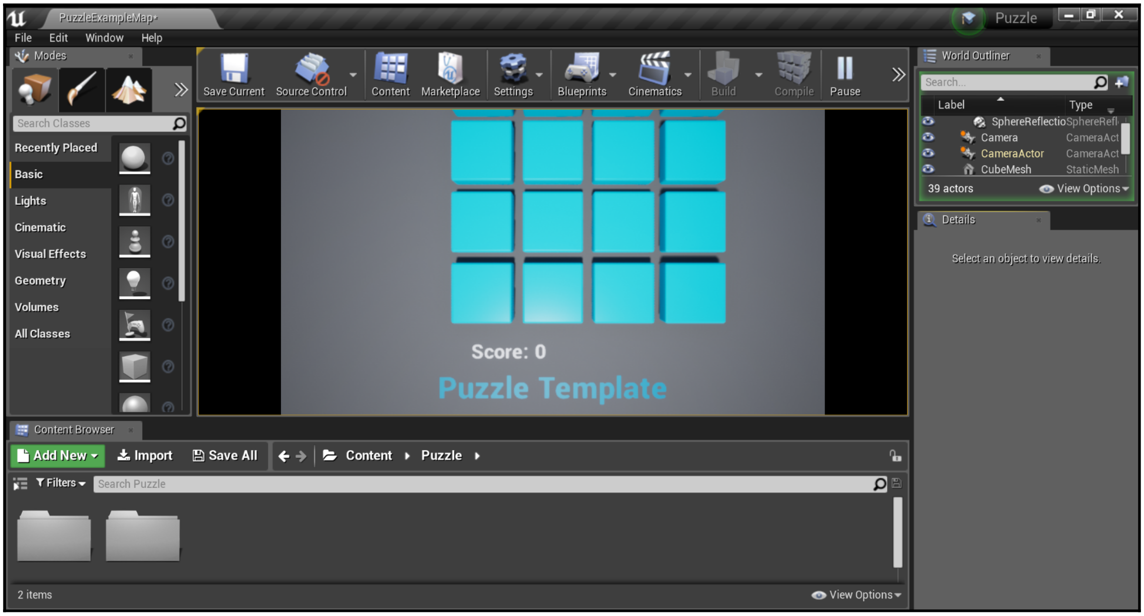
```
void APuzzleBlockGrid::BeginPlay()
```

Notice that there is a `for` loop here to spawn the initial nine blocks, as shown in the following code:

```
// Loop to spawn each block
for( int32 BlockIndex=0; BlockIndex < NumBlocks; BlockIndex++ )
{
  // ...
}
```

Since `NumBlocks` (which is used to determine when to stop the loop) gets computed as `Size*Size`, we can easily change the number of blocks that get spawned by altering the value of the `Size` variable. Go to Line 24 of `PuzzleBlockGrid.cpp` and change the value of the `Size` variable to `4` or `5`. Then, run the code again (make sure you press the **Compile** button in the Unreal Editor to make it use the updated code).

You should see the number of blocks on the screen increase (although you may need to scroll to see them all), as shown in the following screenshot:



Setting the size to `14` creates many more blocks.

Summary

In this chapter, you learned how to repeat lines of code by looping the code, which allowed you to run it multiple times. This can be used to use the same line of code repeatedly in order to achieve a task. Imagine printing the numbers from `1` to `10` (or 10,000!) without using a loop.

In the next chapter, we will explore functions, which are the basic units of reusable code.

5
Functions and Macros

When writing code, you'll find yourself needing to run the same code multiple times. The last thing you want to do is to copy and paste the same code in a bunch of different places (after all, what happens if you need to make a change?). Wouldn't it be easier to just write it once and call it multiple times? That's what we're covering in this chapter. The topics we will cover include the following:

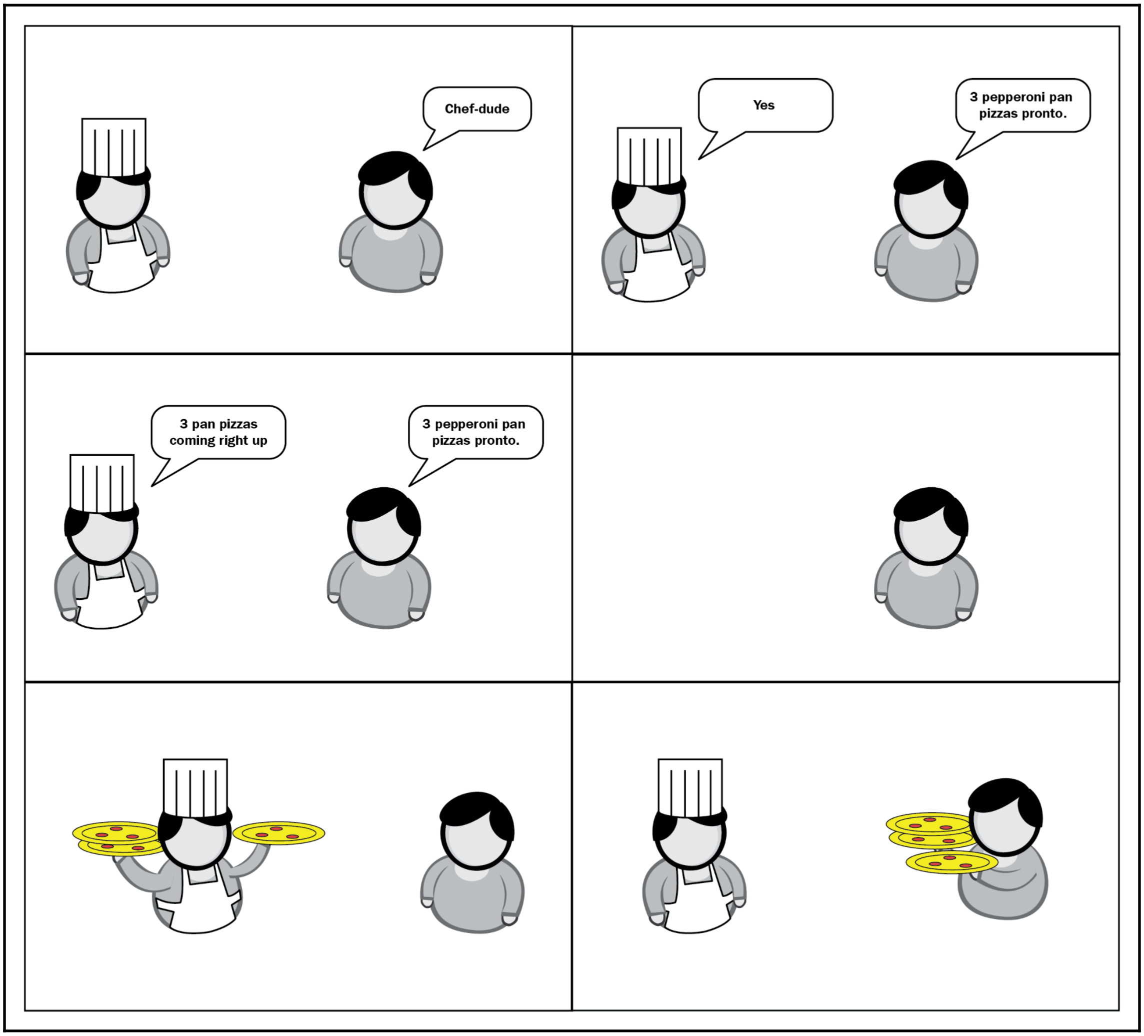
- Functions
- Functions with arguments
- Functions that return a value
- Initializer lists
- More on variables
- Macros
- Constexpr

Functions

Some things need to be repeated. Code is not one of them. A function is a bundle of code that can get called any number of times, as often you wish.

Analogies are good. Let's explore an analogy that deals with waiters, chefs, pizzas, and functions. In English, when we say a person has a function, we mean that the person performs some very specific (usually very important) task. They can do this task again and again, whenever they are called upon to do so.

The following comic strip shows the interaction between a waiter (caller) and a chef (callee). The waiter wants food for his table, so he calls upon the chef to prepare the food required by the waiting table. The chef prepares the food and then returns the result to the waiter:



Here, the chef performs his function of cooking food. The chef accepted the parameters about what type of food to cook (three pepperoni pan pizzas). The chef then went away, did some work, and returned with three pizzas. Note that the waiter does not know and does not care about how the chef goes about cooking the pizzas. The chef abstracts away the process of cooking pizzas for the waiter, so cooking a pizza is just a simple, single-line command for the waiter. The waiter just wants his request to be completed and the pizzas returned to him.

When a function (chef) gets called with some arguments (types of pizzas to be prepared), the function performs some actions (preparing the pizzas) and optionally returns a result (the actual finished pizzas).

An example of a library function – sqrt()

Now, let's talk about a more practical example and relate it to the pizza example.

There is a function in the `<cmath>` library called the `sqrt()` function. Let me quickly illustrate its use, as shown in the following code:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  double rootOf5 = sqrt( 5 ); // function call to the sqrt
   function
  cout << rootOf5  << endl;
}
```

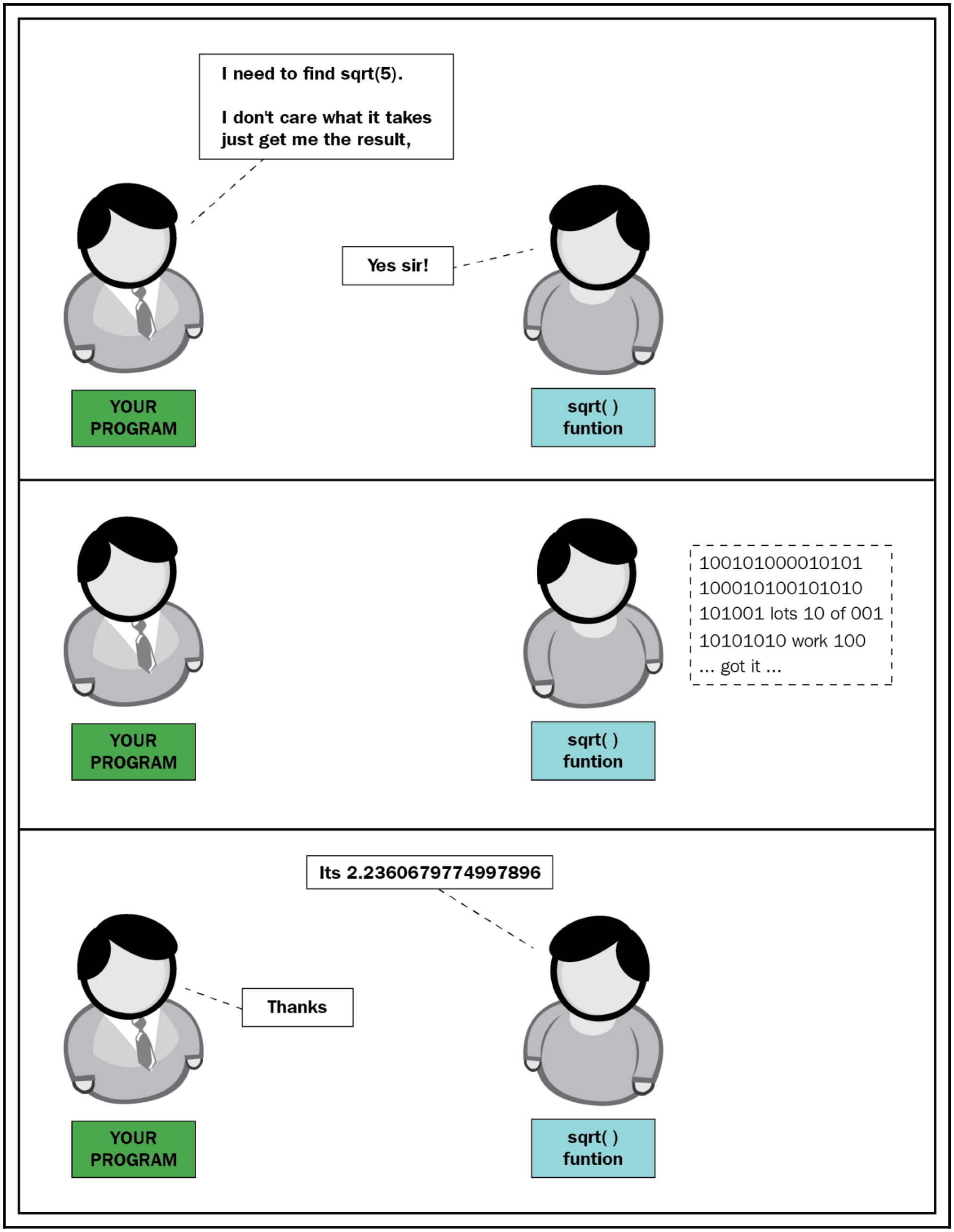
The function call is after the = character: `sqrt( 5 )`. So, `sqrt()` can find the mathematical square root of any number given to it.

Do you know how to find the square root of a tough number such as 5? It's not simple. A clever soul sat down and wrote a function that can find the square roots of all types of numbers. Do you have to understand the math behind how the square root of 5 is found to use the `sqrt(5)` function call? Heck, no! So, just as the waiter didn't have to understand how to cook a pizza in order to get a pizza as the result, the caller of a C++ library function does not have to fully understand how that library function works internally to use it effectively.

The following are the advantages of using functions:

- Functions abstract away a complex task into a simple, callable routine. This makes the code required to *cook a pizza*, for instance, just a single-line command for the caller (the caller is typically your program).
- Functions avoid the repetition of code where it is not necessary. Say we have 20 or so lines of code that can find the square root of a double value. We wrap these lines of code into a callable function; instead of repeatedly copying and pasting these 20 lines of code, we simply call the `sqrt` function (with the number to root) whenever we need a root.

The following diagram shows the process of finding a square root:


I need to find sqrt(5).
I don't care what it takes just get me the result,
Yes sir!
YOUR PROGRAM
sqrt( ) funtion
100101000010101
100010100101010
101001 lots 10 of 001
10101010 work 100
... got it ...
YOUR PROGRAM
sqrt( ) funtion
Its 2.2360679774997896
Thanks
YOUR PROGRAM
sqrt( ) funtion

Writing our own functions

Say we want to write some code that prints out a strip of road, as shown here:

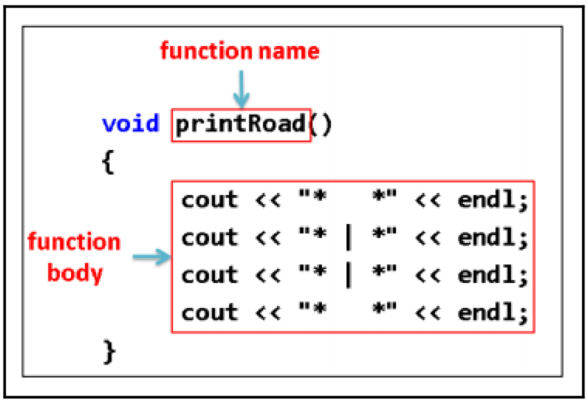
```
cout << "*   *" << endl;
cout << "* | *" << endl;
cout << "* | *" << endl;
cout << "*   *" << endl;
```

Now, say we want to print two strips of road in a row, or three strips of road. Or, say we want to print any number of strips of road. We will have to repeat the four lines of code that produce the first strip of road once per strip of road we're trying to print.

What if we introduced our own C++ command that allowed us to print a strip of road on calling the command? Here's how that will look:

```
void printRoad()
{
  cout << "*   *" << endl;
  cout << "* | *" << endl;
  cout << "* | *" << endl;
  cout << "*   *" << endl;
}
```

This is the definition of a function. A C++ function has the following anatomy:



The `void` means it does not return any values and, since there is nothing inside the parentheses, it doesn't take any parameters. We'll get into parameters and return values later. Using a function is simple: we simply invoke the function we want to execute by name, followed by two round brackets, `()`. For example, calling the `printRoad()` function will cause the `printRoad()` function to run. Let's trace an example program to fully understand what this means.

A sample program trace

Here's a complete example of how a function call works:

```
#include <iostream>
using namespace std;
void printRoad()
{
  cout << "*   *" << endl;
  cout << "* | *" << endl;
  cout << "* | *" << endl;
  cout << "*   *" << endl;
}
int main()
{
  cout << "Program begin!" << endl;
  printRoad();
  cout << "Program end" << endl;
  return 0;
}
```

Let's trace the program's execution from beginning to end. Remember that, for all C++ programs, execution begins on the first line of `main()`.

`main()` is also a function. It oversees the execution of the whole program. Once `main()` executes the `return` statement, your program ends.

A line-by-line trace of the execution of the preceding program is shown as follows:

```
void printRoad()
{
  cout << "*   *" << endl;          // 3: then we jump up here
  cout << "* | *" << endl;          // 4: run this
  cout << "* | *" << endl;          // 5: and this
  cout << "*   *" << endl;          // 6: and this
}
```

```
int main()
{
  cout << "Program begin!" << endl; // 1: first line to execute
  printRoad();                      // 2: second line..
  cout << "Program end" << endl;    // 7: finally, last line
  return 0;                         // 8: and return to o/s
}
```

This is how the output of this program will look:

```
Program begin!
*   *
* | *
* | *
*   *
Program end
```

Here's an explanation of the preceding code, line by line:

1. The program's execution begins on the first line of `main()`, which outputs `program begin!`.
2. The next line of code that is run is the call to `printRoad()`. What this does is it jumps the program counter to the first line of `printRoad()`. All the lines of `printRoad()` then execute in order (lines 3-6).
3. After the function call to `printRoad()` is complete, control returns to the `main()` statement. We then see `Program end` printed.

TIP

Don't forget the brackets after the function call to `printRoad()`. A function call must always be followed by round brackets, `()`, otherwise the function call will not work and you will get a compiler error.

The following code is used to print four strips of road:

```
int main()
{
        printRoad();
        printRoad();
        printRoad();
        printRoad();
}
```

Alternatively, you can also use the following code:

```
for( int i = 0; i < 4; i++ )
{
    printRoad();
}
```

So, instead of repeating the four lines of `cout` every time a box is printed, we simply call the `printRoad()` function to make it print. Also, if we want to change how a printed road looks, we have to simply modify the implementation of the `printRoad()` function.

Calling a function entails running the entire body of that function, line by line. After the function call is complete, the control of the program then resumes at the point of the function call.

Exercise

As an exercise, find out what is wrong with the following code:

```
#include <iostream>
using namespace std;
void myFunction()
{
   cout << "You called?" << endl;
}
int main()
{
   cout << "I'm going to call myFunction now." << endl;
   myFunction;
}
```

Solution

The correct answer to this problem is that the call to `myFunction` (in the last line of `main()`) is not followed by round brackets. All function calls must be followed by round brackets. The last line of `main()` should read `myFunction();`, not just `myFunction`.

Functions with arguments

How can we extend the `printRoad()` function to print a road with a certain number of segments? The answer is simple. We can let the `printRoad()` function accept a parameter, called `numSegments`, to print a certain number of road segments.

The following code snippet shows how that will appear:

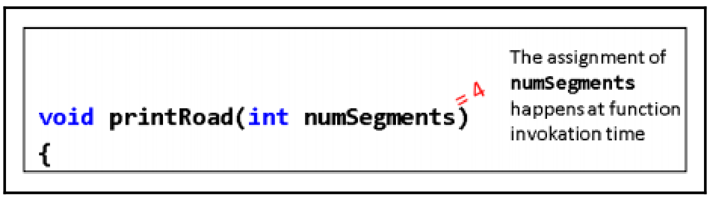
```
void printRoad(int numSegments)
{
  // use a for loop to print numSegments road segments
  for( int i = 0; i < numSegments; i++)
  {
    cout << "*   *" << endl;
    cout << "* | *" << endl;
    cout << "* | *" << endl;
    cout << "*   *" << endl;
  }
}
```

The following screenshot shows the anatomy of a function that accepts an argument:

Call this new version of `printRoad()`, asking it to print four segments, as follows:

```
printRoad( 4 );    // function call
```

The value `4` between the brackets of `function call` in the preceding statement gets assigned to the `numSegments` variable of the `printRoad(int numSegments)` function. This is how the value `4` gets passed to `numSegments`:



An illustration of how printRoad(4) will assign the value 4 to the numSegments variable

So, `numSegments` gets assigned the value passed between the brackets in the call to `printRoad()`.

Functions that return values

An example of a function that returns a value is the `sqrt()` function. The `sqrt()` function accepts a single parameter between its brackets (the number to root) and returns the actual root of the number.

Here's an example using the `sqrt` function:

```
cout << sqrt( 4 ) << endl;
```

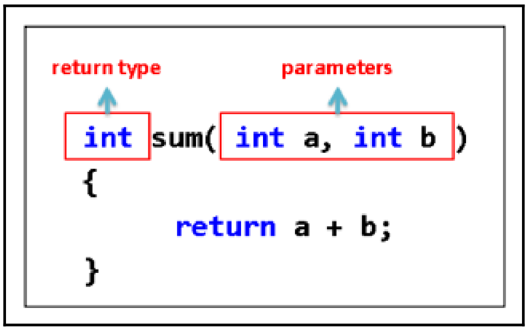
The `sqrt()` function does something analogous to what the chef did when preparing the pizzas.

As a caller of the function, you do not care about what goes on inside the body of the `sqrt()` function; that information is irrelevant since all you want is the result of the square root of the number that you are passing.

Let's declare our own simple function that returns a value, as shown in the following code:

```
int sum(int a, int b)
{
  return a + b;
}
```

The following screenshot shows the anatomy of a function with parameters and a returned value:



The `sum` function is very basic. All it does is take two `int` numbers, `a` and `b`, sums them up together, and returns a result. You might say that we don't even need an entire function just to add two numbers. You're right, but bear with me for a moment. We will use this simple function to explain the concept of returned values.

You will use the `sum` function in this way (from `main()`):

```
int sum( int a, int b )
{
  return a + b;
}
int main()
{
  cout << "The sum of 5 and 6 is " << sum( 5,6 ) << endl;
}
```

For the `cout` command to complete, the `sum( 5,6 )` function call must be evaluated. At the point where the `sum( 5,6 )` function call occurs, the returned value from `sum( 5,6 )` is put right there.

In other words, this is the line of code that `cout` actually sees after evaluating the `sum( 5,6 )` function call:

```
cout << "The sum of 5 and 6 is " << 11 << endl;
```

The returned value from `sum( 5,6 )` is effectively cut and pasted at the point of the function call. A value must always be returned by a function that promises to do so (if the return type of the function is anything other than `void`).

Exercises

1. Write an `isPositive` function that returns `true` when the double parameter passed to it is indeed positive.
2. Complete the following function definition:

```
// function returns true when the magnitude of 'a'
// is equal to the magnitude of 'b' (absolute value)
bool absEqual(int a, int b)
 {
    // to complete this exercise, try to not use
    // cmath library functions
}
```

3. Write a `getGrade()` function that accepts an integer value (marks out of 100) and returns the grade (either `A`, `B`, `C`, `D`, or `F`).
4. A mathematical function is of the form `f(x) = 3x + 4`. Write a C++ function that returns values for `f(x)`.

Solutions

1. The `isPositive` function accepts a double parameter and returns a Boolean value:

```
bool isPositive( double value )
{
  return value > 0;
}
```

2. The following is the completed `absEqual` function:

```
bool absEqual( int a, int b )
{
  // Make a and b positive
  if( a < 0 )
  {
    a = -a;
  }
  if( b < 0 )
  {
    b = -b;
  }
  // now since they're both +ve,
  // we just have to compare equality of a and b together
  return a == b;
}
```

3. The `getGrade()` function is given in the following code:

```
char getGrade( int grade )
{
  if( grade >= 90 )
  {
    return 'A';
  }
  else if( grade >= 80 )
  {
    return 'B';
  }
```

```
    else if( grade >= 70 )
    {
      return 'C';
    }
    else if( grade >= 60 )
    {
      return 'D';
    }
    else
    {
      return 'F';
    }
  }
```

4. This program is a simple one that should entertain you. The origin of the name function in C++ actually came from the math world, as shown in the following code:

```
double f( double x )
{
  return 3*x + 4;
}
```

Initializer lists

Sometimes, you may not know how many items you want to pass to an array. Newer versions of C++ have added a simple way of doing this, an initializer list. This allows you to pass in any number of items inside curly brackets and separated by commas, like this:

```
{ 1, 2, 3, 4 }
```

To set this up, you need to use `initializer_list` as the type:

```
#include <initializer_list>
using namespace std;

int sum(initializer_list<int> list) {
    int total = 0;
    for (int e : list) { // Iterate through the list
        total += e;
    }

    return total;
}
```

This is a template, which we will go into later, but for now all you need to know is the type of object you're putting in the list is inside the angle brackets like this: `<int>`. This could just as easily been another type, such as `float` or `char`.

To call this function, you can pass in the values like this:

```
sum({ 1, 2, 3, 4 });
```

The result will be `10` for this case.

Variables revisited

It's always nice to revisit a topic you've covered before, now that you understand C++ coding in much more depth.

Global variables

Now that we've introduced the concept of functions, the concept of a global variable can be introduced.

What is a global variable? A global variable is any variable that is made accessible to all of the functions of the program. How can we make a variable that is accessible to all of the functions of the program? We simply declare the global variable at the top of the code file, usually after or near the `#include` statements.

Here's an example program with some global variables:

```
#include <iostream>
#include <string>
using namespace std;

string g_string;             // global string variable,
// accessible to all functions within the program
// (because it is declared before any of the functions
// below!)

void addA(){ g_string += "A"; }
void addB(){ g_string += "B"; }
void addC(){ g_string += "C"; }

int main()
{
  addA();
```

```
    addB();
    cout << g_string << endl;
    addC();
    cout << g_string << endl;
  }
```

Here, the same `g_string` global variable is accessible to all four functions in the program (`addA()`, `addB()`, `addC()`, and `main()`). Global variables live for the duration of the program.

People sometimes prefer to prefix global variables with `g_`, but prefixing the variable name with `g_` is not a requirement for a variable to be global.

Local variables

A local variable is a variable that is defined within a block of code. Local variables go out of scope at the end of the block in which they are declared. Some examples will follow in the next section, *The scope of a variable*.

The scope of a variable

The scope of a variable is the area of code where that variable can be used. The scope of any variable is basically the block in which it is defined. We can demonstrate a variable's scope using an example, as shown in the following code:

```
int g_int; // global int, has scope until end of file
void func( int arg )
{
  int fx;
} // </fx> dies, </arg> dies

int main()
{
  int x = 0; // variable <x> has scope starting here..
         // until the end of main()
  if( x == 0 )
  {
    int y;  // variable <y> has scope starting here,
            // until closing brace below
  } // </y> dies
  if( int x2 = x ) // variable <x2> created and set equal to <x>
```

```
    {
      // enter here if x2 was nonzero
    } // </x2> dies

    for( int c = 0; c < 5; c++ ) // c is created and has
    { // scope inside the curly braces of the for loop
      cout << c << endl;
    } // </c> dies only when we exit the loop
  } // </x> dies
```

The main thing that defines a variable's scope is a block. Let's discuss the scope of a couple of the variables defined in the preceding code example:

- `g_int`: This is a global integer with a scope that ranges from the point it was declared until the end of the code file. That is to say, `g_int` can be used inside `func()` and `main()`, but it cannot be used in other code files. To have a single global variable that is used across multiple code files, you will need an external variable.
- `arg` (the argument of `func()`): This can be used from the first line of `func()` (after the opening curly brace, `{`) to the last line of `func()` (until the closing curly brace, `}`).
- `fx`: This can be used anywhere inside `func()` until the closing curly brace (`}`) of `func()`.
- `main()` (variables inside `main()`): This can be used as marked in the comments.

Notice how variables declared inside the brackets of a function's argument list can only be used inside the block below that function's declaration, for example, the `arg` variable passed to `func()`:

```
void func( int arg )
{
  int fx;
} // </fx> dies, </arg> dies
```

The `arg` variable will die after the closing curly brace (`}`) of the `func()` function. This is counter intuitive as the round brackets are technically outside the curly braces that define the `{` block `}`.

The same goes for variables declared inside the round brackets of a `for` loop. Take as an example the following `for` loop:

```
for( int c = 0; c < 5; c++ )
{
  cout << c << endl;
} // c dies here
```

The `int c` variable can be used inside the round brackets of the `for` loop declaration or inside the block below the `for` loop declaration. The `c` variable will die after the closing of the curly brace of the `for` loop it is declared in. If you want the `c` variable to live on after the brace brackets of the `for` loop, you need to declare the `c` variable before the `for` loop, as shown here:

```
int c;
for( c = 0; c < 5; c++ )
{
  cout << c << endl;
} // c does not die here
```

Static local variables

`static` local variables have a local scope, but they don't go away when you exit the function, and instead remember the value between calls, as shown in the following code:

```
void testFunc()
{
  static int runCount = 0; // this only runs ONCE, even on
  // subsequent calls to testFunc()!
  cout << "Ran this function " << ++runCount << " times" << endl;
} // runCount stops being in scope, but does not die here

int main()
{
  testFunc();  // says 1 time
  testFunc();  // says 2 times!
}
```

With the use of the `static` keyword inside `testFunc()`, the `runCount` variable remembers its value between calls of `testFunc()`. So, the output of the two separate preceding runs of `testFunc()` is as follows:

```
Ran this function 1 times
Ran this function 2 times
```

That's because static variables are only created and initialized once (the first time when the function they are declared in runs), and after that, the static variable retains its old value. Say we declare `runCount` as a regular, local, nonstatic variable:

```
int runCount = 0; // if declared this way, runCount is local
```

Then, this is how the output will look:

```
Ran this function 1 times
Ran this function 1 times
```

Here, we see `testFunc` saying `Ran this function 1 time` both times. As a local variable, the value of `runCount` is not retained between function calls.

You should not overuse static local variables. In general, you should only use a static local variable when it is absolutely necessary.

Const variables

A `const` variable is a variable whose value you promise the compiler not to change after the first initialization. We can declare one simply, for example, for the value of `pi`:

```
const double pi = 3.14159;
```

Since `pi` is a universal constant (one of the few things you can rely on to be the same), there should be no need to change `pi` after initialization. In fact, changes to `pi` should be forbidden by the compiler. Try, for example, to assign `pi` a new value:

```
pi *= 2;
```

We will get the following compiler error:

```
error C3892: 'pi' : you cannot assign to a variable that is const
```

This error makes perfect sense because, besides the initialization, we should not be able to change the value of `pi`—a variable that is constant.

Const and functions

`const` can be used in many ways, some of which involve functions. Sometimes, you are passing a variable into a function, but you don't want the function to make any changes to the value. You may think well, I can just make sure I don't change it, can't I? That may be the case on your own projects, but what if you're on a big team with multiple programmers? You could just put a comment, but it's generally better to make sure the parameter is marked as `const`. To do that, you write the function like this:

```
int sum(const int x, const int y)
{
    return x + y;
}
```

Now, if you try to change either of these values, you will cause an error. For example, this won't work:

```
int sum(const int x, const int y)
{
    x = x + y; //ERROR!
    return x;
}
```

You can also return a constant value by changing it to something like this:

```
const int returnConst()
```

Just make sure you save the value that the function returns in a variable that is also marked as `const` or you'll get an error.

Function prototypes

A function prototype is the signature of the function without the body. For example, let's prototype the `isPositive`, `absEqual`, and `getGrade` functions from the following exercises:

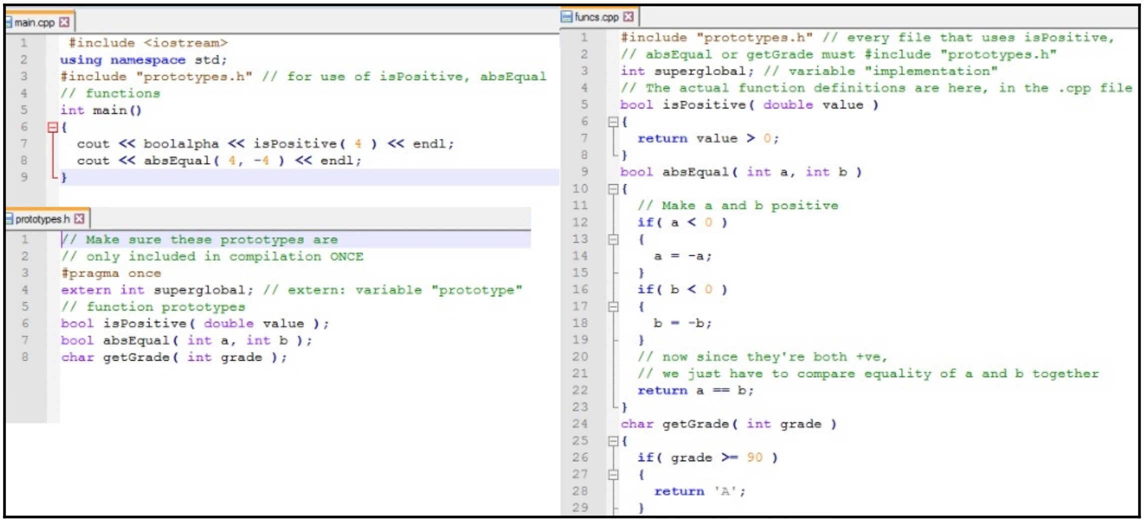
```
bool isPositive( double value );
bool absEqual( int a, int b );
char getGrade( int grade );
```

Notice how the function prototypes are just the return type, function name, and argument list that the function requires. Function prototypes don't get a body. The body of the function is typically put in the `.cpp` file.

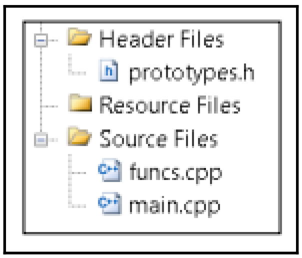
.h and .cpp files

It is typical to put your function prototypes in a `.h` file and the bodies of the functions in a `.cpp` file. The reason for this is you can include your `.h` file in a bunch of `.cpp` files and not get multiple definition errors.

The following screenshot gives you a clear picture of `.h` and `.cpp` files, showing `.cpp` files for the main code and for functions, and a `.h` file holding the function prototypes:



Here, we have three files in this Visual C++ project:



prototypes.h

The `prototypes.h` file contains function prototypes. We will explain what the `extern` keyword does later:

```
// Make sure these prototypes are
// only included in compilation ONCE
#pragma once
extern int superglobal; // extern: variable "prototype"
// function prototypes
bool isPositive( double value );
bool absEqual( int a, int b );
char getGrade( int grade );
```

funcs.cpp

The following is the content of `funcs.cpp`:

```
#include "prototypes.h" // every file that uses isPositive,
// absEqual or getGrade must #include "prototypes.h"
int superglobal; // variable "implementation"
// The actual function definitions are here, in the .cpp file
bool isPositive( double value )
{
  return value > 0;
}
bool absEqual( int a, int b )
{
  // Make a and b positive
  if( a < 0 )
  {
    a = -a;
  }
  if( b < 0 )
  {
    b = -b;
  }
  // now since they're both +ve,
  // we just have to compare equality of a and b together
  return a == b;
}
char getGrade( int grade )
{
  if( grade >= 90 )
  {
    return 'A';
```

```
    }
    else if( grade >= 80 )
    {
      return 'B';
    }
    else if( grade >= 70 )
    {
      return 'C';
    }
    else if( grade >= 60 )
    {
      return 'D';
    }
    else
    {
      return 'F';
    }
  }
```

main.cpp

The following is the content of `main.cpp`:

```
 #include <iostream>
using namespace std;
#include "prototypes.h" // for use of isPositive, absEqual
// functions
int main()
{
  cout << boolalpha << isPositive( 4 ) << endl;
  cout << absEqual( 4, -4 ) << endl;
}
```

When you split up the code into `.h` and `.cpp` files, the `.h` file (the header file) is called the interface and the `.cpp` file (the one with the actual functions in it) is called the implementation.

The puzzling part at first for some programmers is how does C++ know where the `isPositive` and `getGrade` function bodies are, if we only `#include` the prototypes? Shouldn't we `#include` the `funcs.cpp` file into `main.cpp` too?

The answer is *magic*. You only need to `#include` the `prototypes.h` header file in both `main.cpp` and `funcs.cpp`. As long as both `.cpp` files are included in your C++ **Integrated Development Environment** (**IDE**) project (that is, they appear in the **Solution Explorer** tree view at the left-hand side), the linkup of the prototypes to the function bodies is done automatically by the compiler.

extern variables

An `extern` declaration is similar to a function prototype, only it is used on a variable. You can put an `extern` global variable declaration in a `.h` file and include this `.h` file in a whole bunch of other files. This way, you can have a single global variable that gets shared across multiple source files, without getting the multiply defined symbols found linker error. You'd put the actual variable declaration in a `.cpp` file so that the variable only gets declared once. There is an `extern` variable in the `prototypes.h` file in the previous example.

Macros

C++ macros are from a class of C++ commands called preprocessor directives. A preprocessor directive is executed before compilation takes place. Macros start with `#define`. For example, say we have the following macro:

```
#define PI 3.14159
```

At the lowest level, macros are simply copy and paste operations that occur just before compile time. In the preceding macro statement, the `3.14159` literal will be copied and pasted everywhere the symbol `PI` occurs in the program.

Take as an example the following code:

```
#include <iostream>
using namespace std;
#define PI 3.14159
int main()
{
  double r = 4;
  cout << "Circumference is " << 2*PI*r << endl;
}
```

What the C++ preprocessor will do is first go through the code and look for any use of the `PI` symbol. It will find one such use on this line:

```
cout << "Circumference is " << 2*PI*r << endl;
```

The preceding line will convert into the following just before compilation:

```
cout << "Circumference is " << 2*3.14159*r << endl;
```

So, all that happens with a `#define` statement is that all of the occurrences of the symbol used (for example, `PI`) are replaced by the literal number `3.14159` even before compilation occurs. The point of using macros in this way is to avoid hardcoding numbers into the code. Symbols are typically easier to read than big, long numbers.

TIP

Advice: try to use `const` variables where possible.

You can use macros to define constant variables. You can also use `const` variable expressions instead. So, say we have the following line of code:

```
#define PI 3.14159
```

We will be encouraged to use the following instead:

```
const double PI = 3.14159;
```

Using a `const` variable will be encouraged because it stores your value inside an actual variable. The variable is typed and typed data is a good thing.

Macros with arguments

We can also write macros that accept arguments. Here's an example of a macro with an argument:

```
#define println(X) cout << X << endl;
```

What this macro will do is every time `println("Some value")` is encountered in the code, the code on the right-hand side (`cout << "Some value" << endl`) will be copied and pasted into the console. Notice how the argument between the brackets is copied in the place of `X`. Say we had the following line of code:

```
println( "Hello there" )
```

This will be replaced by the following statement:

```
cout << "Hello there" << endl;
```

Macros with arguments are exactly like very short functions. Macros cannot contain any newline characters in them.

Advice: use inline functions instead of macros with arguments.

You have to know about how macros with arguments work because you will encounter them in C++ code a lot. Whenever possible, however, many C++ programmers prefer to use inline functions over macros with arguments.

A normal function call execution involves a `jump` instruction to the function and then the execution of the function. An inline function is one whose lines of code are copied to the point of the function call and no jump is issued. Using inline functions usually makes sense for very small, simple functions that don't have a lot of lines of code. For example, we might inline a simple function, `max`, that finds the larger of two values:

```
inline int max( int a, int b )
{
  if( a > b ) return a;
  else return b;
}
```

Everywhere this `max` function is used, the code for the function body will be copied and pasted at the point of the function call. Not having to `jump` to the function saves execution time, making inline functions effectively similar to macros.

There is a catch to using inline functions. Inline functions must have their bodies completely contained in the `.h` header file. This is so that the compiler can make optimizations and actually inline the function wherever it is used. Functions are made inline typically for speed (since you don't have to jump to another body of the code to execute the function) but at the cost of code bloat.

The following are the reasons why inline functions are preferred over macros:

- Macros are error prone: The argument to the macro is not typed.
- Macros have to be written in one line or you will see them using escaped:

```
 \
newline characters \
like this \
which is hard to read \
```

- If the macro is not carefully written, it will result in difficult-to-fix compiler errors. For example, if you do not bracket your argument properly, your code will just be wrong.
- Large macros are hard to debug.

It should be said that macros do allow you to perform some preprocessor compilation magic. UE4 makes a lot of use of macros with arguments, as you will see later.

Constexpr

There is one other new way you can also do things that happen at compile time, instead of at runtime, and that is by using `constexpr`. As with macros, you can create variables and functions that will get automatically copied by the compiler to where they are used. So, you can do variables like this:

```
constexpr float pi = 3.14129f;
```

You can also add `constexpr` to functions you want to run at compile time like this:

```
constexpr int increment(int i)
{
    return i + 1;
}
```

One more thing you can do with `constexpr` is use it with `if` statements to evaluate something at compile time. So, if you want to do something different for the demo version of the game when you compile it, you can do something like this:

```
if constexpr (kIsDemoVersion) {
    //use demo version code here
} else {
   //use regular version code here
}
```

You'll find more uses for these when we talk about templates.

Summary

Function calls allow you to reuse basic code. Code reuse is important for a number of reasons, mainly because programming is hard and duplicating effort should be avoided wherever possible. The efforts of the programmer that wrote the `sqrt()` function do not need to be repeated by other programmers who want to solve the same problem.

6
Objects, Classes, and Inheritance

In the previous chapter, we discussed functions as a way to bundle up a bunch of lines of related code. We talked about how functions abstract away implementation details and how the `sqrt()` function does not require you to understand how it works internally to use it to find roots. This is a good thing, primarily because it saves programmers time and effort, while making the actual work of finding square roots easier. This principle of abstraction will come up again here, when we discuss objects.

In this chapter, we will be covering:

- What is an object?
- Structs
- Class versus struct
- Getters and setters
- Constructors and destructors
- Class inheritance
- Multiple inheritance
- Putting your class into headers
- Object oriented programming design patterns
- Callable objects and invoke

This chapter contains a lot of keywords that might be difficult to grasp at first, including `virtual` and `abstract`.

Don't let the more difficult sections of this chapter bog you down. I included descriptions of many advanced concepts for completeness. However, bear in mind that you don't need to completely understand everything in this chapter to write working C++ code in UE4. It helps to understand everything, but if something doesn't make sense, don't get stuck. Give it a read and then move on. Probably what will happen is you will not get it at first, but remember a reference to the concept in question when you're coding. Then, when you open this book up again, voilà! It will make sense.

What is an object?

In a nutshell, objects tie together methods (another word for functions) and their related data into a single structure. This structure is called a class. The main idea behind using objects is to create a code representation for everything inside your game. Every object represented in the code will have data and associated functions that operate on that data. So, you'd have an object to represent your `Player` and related functions that make the `Player` `jump()`, `shoot()`, and `pickupItem()`. You'd also have an object to represent every monster instance and related functions, such as `growl()`, `attack()`, and possibly `follow()`.

Objects are types of variables, though, and objects will stay in memory as long as you keep them there. You create an instance, or a specific representation of an object with it's own set of values, once when the thing in your game it represents is created, and you destroy the object instance when the thing in your game it represents dies.

Objects can be used to represent in-game things, but they can also be used to represent any other type of thing. For example, you can store an image as an object. The data fields will be the image's width of the image, its height, and the collection of pixels inside it. C++ strings are also objects.

The struct object

An object in C++ is basically any variable type that is made up of a conglomerate of simpler types. The most basic object in C++ is `struct`. We use the `struct` keyword to glue together a bunch of smaller variables into one big variable. If you recall, we did introduce `struct` briefly in `Chapter 2`, *Variables and Memory*. Let's revise that simple example:

```
struct Player
{
  string name;
  int hp;
};
```

This is the structure definition for what makes a `Player` object. The `Player` has a `string` for their `name` and an integer for their `hp` value.

If you recall from `Chapter 2`, *Variables and Memory*, the way we make an instance of the `Player` object is like this:

```
Player me;    // create an instance of Player, called me
```

From here, we can access the fields of the `me` object like so:

```
me.name = "Tom";
me.hp = 100;
```

Member functions

Now, here's the exciting part. We can attach member functions to the `struct` definition simply by writing these functions inside the `struct Player` definition:

```
struct Player
{
  string name;
  int hp;
  // A member function that reduces player hp by some amount
  void damage( int amount )
  {
    hp -= amount;
  }
  void recover( int amount )
  {
    hp += amount;
  }
};
```

A member function is just a C++ function that is declared inside a `struct` or `class` definition.

There is a bit of a funny idea here, so I'll just come out and say it. The variables of `struct Player` are accessible to all the functions inside `struct Player`. Inside each of the member functions of `struct Player`, we can actually access the `name` and `hp` variables as if they were local to the function. In other words, the `name` and `hp` variables of `struct Player` are shared between all the member functions of `struct Player`.

The this keyword

In some C++ code (in later chapters), you will see more references to the `this` keyword. The `this` keyword is a pointer that refers to the current object. Inside the `Player::damage()` function, for example, we can write our reference to `this` explicitly:

```
void damage( int amount )
{
  this->hp -= amount;
}
```

The `this` keyword only makes sense inside a member function. We could explicitly include the use of the `this` keyword inside member functions, but without writing `this`, it is implied that we are talking about the `hp` of the current object. So, while this is not strictly necessary in most cases, it may be a personal or company preference and could make the code more readable.

Are strings objects?

Yes, strings are objects! Every time you've used a `string` variable in the past, you were using an object. Let's try out some of the member functions of the `string` class.

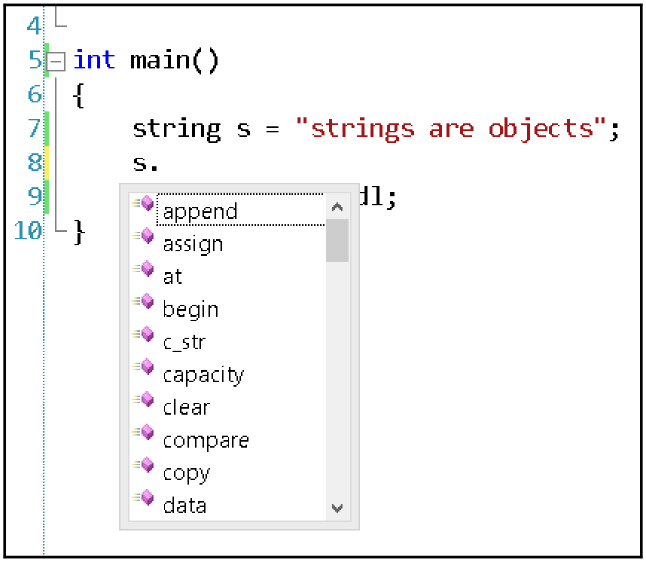
```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string s = "strings are objects";
  s.append( "!!" ); // add on "!!" to end of the string!
  cout << s << endl;
}
```

What we've done here is use the `append()` member function to add on two extra characters to the end of the string (`!!`). Member functions always apply to the object that calls the member function (the object to the left of the dot).

To see the listing of members and member functions available on an object, follow these steps:

1. Type the object's variable name in Visual Studio
2. Then type a dot (`.`)
3. Then press *Ctrl* and the spacebar

A member listing will pop up as follows:



Pressing Ctrl and the spacebar will make the member listing appear

Invoking a member function

Member functions can be invoked with the following syntax:

```
objectName.memberFunction();
```

The object invoking the member function is on the left of the dot. The member function to call is on the right of the dot. A member function invocation is always followed by round brackets `()`, even when no arguments are passed to the brackets.

So, in the part of the program where the monster attacks, we can reduce the `player`'s `hp` value as follows:

```
player.damage( 15 );  // player takes 15 damage
```

Isn't that more readable than the following?

```
player.hp -= 15;      // player takes 15 damage
```

When member functions and objects are used effectively, your code will read more like prose or poetry than a bunch of operator symbols slammed together.

Besides beauty and readability, what is the point of writing member functions? Outside the `Player` object, we can now do more with a single line of code than just reduce the `hp` member by `15`. We can also do other things as we're reducing the `player`'s `hp`, such as take into account the `player`'s armor, check whether the player is invulnerable, or have other effects occur when the `Player` is damaged. What happens when the player is damaged should be abstracted away by the `damage()` function.

Now, imagine the `Player` had an `armorClass`. Let's add a field to `struct Player` for `armorClass`:

```
struct Player
{
  string name;
  int hp;
  int armorClass;
};
```

We'd need to reduce the damage received by the `Player` by the armor class of the `Player`. So, we'd type a formula to reduce `hp`. We can do it the non-object-oriented way by accessing the data fields of the `Player` object directly:

```
player.hp -= 15 - player.armorClass; // non OOP
```

Otherwise, we can do it the object-oriented way by writing a member function that changes the data members of the `Player` object as needed. Inside the `Player` object, we can write a `damage()` member function:

```
struct Player
{
  string name;
  int hp;
  int armorClass;
  void damage( int dmgAmount )
  {
    hp -= dmgAmount - armorClass;
  }
};
```

Exercises

1. There is a subtle bug in the `Player`'s `damage` function in the preceding code. Can you find and fix it? Hint: what happens if the damage dealt is less than the `armorClass` of the `Player`?
2. Having only a number for armor class doesn't give enough information about the armor! What is the armor's name? What does it look like? Devise a `struct` function for the `Player`'s armor with fields for `name`, `armorClass`, and `durability` rating.

Solutions

The solution to the first exercise is in the `struct Player` code listed in the next section, *Privates and encapsulation*.

For the second, how about using the following code?

```
struct Armor
{
  string name;
  int armorClass;
  double durability;
};
```

An instance of `Armor` will then be placed inside `struct Player`:

```
struct Player
{
  string name;
  int hp;
  Armor armor; // Player has-an Armor
};
```

This means the `Player` has an armor. Keep this in mind—we'll explore `has-a` versus `is-a` relationships later.

> **TIP:** All variable names thus far start with a lowercase character. This is a good convention with C++ code. You may find some cases where specific teams or other languages prefer to use uppercase characters to start variable names, in which case it's better to just do what people at your company expect.

Privates and encapsulation

So now we've defined a couple of member functions, whose purpose it is to modify and maintain the data members of our `Player` object, but some people have come up with an argument.

The argument is as follows:

- An object's data members should only ever be accessed through its member functions, never directly.

  This means that you should never access an object's data members from outside the object directly, in other words, modify the `player`'s `hp` directly:

  ```
  player.hp -= 15 - player.armorClass; // bad: direct member access
  ```

  This should be forbidden, and users of the class should be forced to use the proper member functions to change the values of data members instead:

  ```
  player.damage( 15 );  // right: access through member function
  ```

  This principle is called *encapsulation*. Encapsulation is the concept that every object should be interacted via its member functions only. Encapsulation says that raw data members should never be accessed directly.

The reasons behind encapsulation are as follows:

- **To make the class self-contained**: The primary idea behind encapsulation is that objects work best when they are programmed such that they manage and maintain their own internal state variables without the need for code outside the class to examine that class's private data. When objects are coded this way, it makes the object much easier to work with, that is, easier to read and maintain. To make the `Player` object jump, you should just have to call `player.jump();` let the `Player` object manage state changes to its `y-height` position (making the `Player` jump!). When an object's internal members are not exposed, interacting with that object is much easier and more efficient. Interact only with an object's public member functions; let the object manage its internal state (we will explain the keywords `private` and `public` in a moment).
- **To avoid breaking code**: When code outside of a class interacts with that class's public member functions only (the class's public interface), then an object's internal state management is free to change, without breaking any of the calling code. This way, if an object's internal data members change for any reason, all code using the object still remains valid, as long as the member functions' signatures - the names, return types, and any parameters—remain the same.

So, how can we prevent a programmer from doing the wrong thing and accessing data members directly? C++ introduces the concept of *access modifiers* to prevent accessing an object's internal data.

Here is how we'd use access modifiers to forbid access to certain sections of `struct Player` from outside of `struct Player`.

The first thing you'd do is decide which sections of the `struct` definition you want to be accessible outside of the class. These sections will be labelled `public`. All other regions that will not be accessible outside of `struct` will be labelled `private`,
as follows:

```
struct Player
{
private:        // begins private section.. cannot be accessed
                // outside the class until
  string name;
  int hp;
  int armorClass;
public:         //  until HERE. This begins the public section
  // This member function is accessible outside the struct
  // because it is in the section marked public:
  void damage( int amount )
```

```
    {
      int reduction = amount - armorClass;
      if( reduction < 0 ) // make sure non-negative!
        reduction = 0;
      hp -= reduction;
    }
  };
```

Some people like it public

Some people do unabashedly use `public` data members and do not encapsulate their objects. This is a matter of preference, though is considered bad object-oriented programming practice.

However, classes in UE4 do use `public` members sometimes. It's a judgment call; whether a data member should be `public` or `private` is really up to the programmer.

With experience, you will find that, sometimes, you get into a situation that requires quite a bit of refactoring (modifying code) when you make a data member `public` that should have been `private`.

The class keyword versus struct

You might have seen a different way of declaring an object, using the `class` keyword, instead of `struct`, as shown in the following code:

```
class Player // we used class here instead of struct!
{
  string name;
  //
};
```

The `class` and `struct` keywords in C++ are almost identical. There is only one difference between `class` and `struct`, and that is that the data members inside a `struct` keyword will be declared `public` by default, while in a `class` keyword, the data members inside the class will be declared `private` by default. (This is why I introduced objects using `struct`; I didn't want to inexplicably put `public` as the first line of `class`.)

In general, `struct` is preferred for simple types that don't use encapsulation, don't have many member functions, and must be backward-compatible with C. Classes are used almost everywhere else.

From now on, let's use the `class` keyword instead of `struct`.

Getters and setters

You might have noticed that, once we slap `private` onto the `Player` class definition, we can no longer read or write the name of the `Player` from outside the `Player` class.

Say we try and read the name with the following code:

```
Player me;
cout << me.name << endl;
```

Or write to the name, as follows:

```
me.name = "William";
```

Using the `struct Player` definition with `private` members, we will get the following error:

```
main.cpp(24) : error C2248: 'Player::name' : cannot access private
member declared in class 'Player'
```

This is just what we asked for when we labeled the `name` field `private`. We made it completely inaccessible outside the `Player` class.

Getters

A getter (also known as an accessor function) is used to pass back copies of internal data members to the caller. To read the `Player`'s name, we'd deck out the `Player` class with a member function, specifically to retrieve a copy of that `private` data member:

```
class Player
{
private:
  string name;  // inaccessible outside this class!
                //  rest of class as before
public:
  // A getter function retrieves a copy of a variable for you
  string getName()
  {
    return name;
  }
};
```

So, now it is possible to read the `player`'s `name` information. We can do this by using the following code statement:

```
cout << player.getName() << endl;
```

Getters are used to retrieve `private` members that would otherwise be inaccessible to you from outside the class.

Real world tip - the const keyword

Inside a class, you can add the `const` keyword to a member function declaration. What the `const` keyword does is promise to the compiler that the internal state of the object will not change as a result of running this function. Attaching the `const` keyword will look something like this:

```
string getName() const
{
 return name;
}
```

No assignments to data members can happen inside a member function that is marked `const`. As the internal state of the object is guaranteed not to change as a result of running a `const` function, the compiler can make some optimizations regarding function calls to `const` member functions.

Setters

A setter (also known as a modifier function or mutator function) is a member function whose sole purpose is to change the value of an internal variable inside the class, as shown in the following code:

```
class Player
{
private:
  string name;  // inaccessible outside this class!
                //  rest of class as before
public:
  // A getter function retrieves a copy of a variable for you
  string getName()
  {
    return name;
  }
  void setName( string newName )
```

```
    {
      name = newName;
    }
  };
```

So we can still change a `private` variable in a `class` from outside the `class` function, but only if we do so through a setter function.

But what's the point of get/set operations?

So, the first question that crosses a newbie programmer's mind when they first encounter get/set operations on `private` members is, isn't get / set self-defeating? I mean, what's the point in hiding access to data members when we're just going to expose that same data again in another way? It's like saying, *"You can't have any chocolates because they are private, unless you say please* `getMeTheChocolate()`*. Then, you can have the chocolates."*

Some expert programmers even shorten the get/set functions to one-liners, like this:

```
string getName(){ return name; }
void setName( string newName ){ name = newName; }
```

Let's answer the question. Doesn't a get/set pair break encapsulation by exposing the data completely?

The answer is twofold. First, get member functions typically only return a copy of the data member being accessed. This means that the original data member's value remains protected and is not modifiable through a `get()` operation.

A `set()` (mutator method) operation is a little bit counter-intuitive though. If the setter is a `passthru` operation, such as `void setName( string newName ) { name=newName; }`, then having the setter might seem pointless. What is the advantage of using a mutator method instead of overwriting the variable directly?

The argument for using mutator methods is to write additional code before the assignment of a variable to guard the variable from taking on incorrect values.
Say, for example, we have a setter for the `hp` data member, which will look like this:

```
void setHp( int newHp )
{
  // guard the hp variable from taking on negative values
  if( newHp < 0 )
  {
    cout << "Error, player hp cannot be less than 0" << endl;
    newHp = 0;
```

```
    }
    hp = newHp;
  }
```

The mutator method is supposed to prevent the internal `hp` data member from taking on negative values. You might consider mutator methods a bit retroactive. Should the responsibility lie with the calling code to check the value it is setting before calling `setHp( -2 )`, and not let that only get caught in the mutator method? Can't you use a `public` member variable and put the responsibility for making sure the variable doesn't take on invalid values in the calling code, instead of in the setter? You can.

This is the core reason behind using mutator methods. The idea behind mutator methods is that the calling code can pass any value it wants to the `setHp` function (for example, `setHp( -2 )`), without having to worry whether the value it is passing to the function is valid or not. The `setHp` function then takes the responsibility of ensuring that the value is valid for the `hp` variable.

Some programmers consider direct mutator functions such as `getHp()`/`setHp()` a code smell. A code smell is, in general, a bad programming practice that people don't overtly take notice of, except for a niggling feeling that something is being done sub-optimally. They argue that higher-level member functions can be written instead of mutators. For example, instead of a `setHp()` member function, we should have `public` member functions such as `heal()` and `damage()` instead. An article on this topic is available at `http://c2.com/cgi/wiki?AccessorsAreEvil`.

Constructors and destructors

The constructor in your C++ code is a simple little function that runs once when the C++ object instance is first created. The destructor runs once when the C++ object instance is destroyed. Say we have the following program:

```
#include <iostream>
#include <string>
using namespace std;
class Player
{
private:
  string name;  // inaccessible outside this class!
public:
  string getName(){ return name; }
// The constructor!
  Player()
  {
```

```
    cout << "Player object constructed" << endl;
    name = "Diplo";
  }
  // ~Destructor (~ is not a typo!)
  ~Player()
  {
    cout << "Player object destroyed" << endl;
  }
};

int main()
  {
    Player player;
    cout << "Player named '" << player.getName() << "'" << endl;
  }
  // player object destroyed here
```

Here, we have created a `Player` object. The output of this code will be as follows:

```
Player object constructed
Player named 'Diplo'
Player object destroyed
```

The first thing that happens during object construction is that the constructor actually runs. This prints the line `Player object constructed`. Following this, the line with the `Player`'s name gets printed: `Player named 'Diplo'`. Why is the `Player` named Diplo? Because that is the name assigned in the `Player()` constructor.

Finally, at the end of the program, the `Player` destructor gets called, and we see `Player object destroyed`. The `Player` object gets destroyed when it goes out of scope at the end of `main()` (at `}` of `main`).

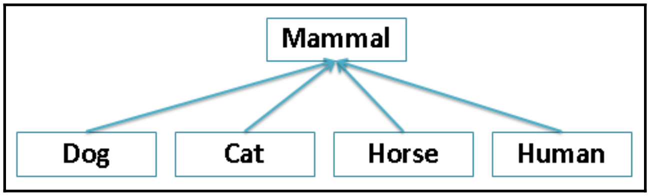
So, what are constructors and destructors good for? Exactly what they appear to be for: setting up and tearing down an object. The constructor can be used for the initialization of data fields, and the destructor to call `delete` on any dynamically allocated resources (we haven't covered dynamically allocated resources yet, so don't worry about this last point).

Class inheritance

You use inheritance when you want to create a new, more functional class of code, based on an existing class of code. Inheritance is a tricky topic to cover. Let's start with the concept of a derived class (or subclass).

Derived classes

The most natural way to consider inheritance is by analogy with the animal kingdom. The classification of living things is shown in the following diagram:



What this diagram means is that **Dog**, **Cat**, **Horse**, and **Human** are all mammals. What that means is that they all all share some common characteristics, such as having common organs (a brain with a neocortex, lungs, a liver, and a uterus in females), while being completely different in other regards. How each walks is different. How each communicates is also different.

What would that mean if you were coding creatures? You would only have to program the common functionality once. Then, you would implement the code for the different parts specifically for each of the `Dog`, `Cat`, `Horse`, and `Human` classes.

A concrete example of the preceding diagram is as follows:

```
#include <iostream>
using namespace std;
class Mammal
{
protected:
  // protected variables are like privates: they are
  // accessible in this class but not outside the class.
  // the difference between protected and private is
  // protected means accessible in derived subclasses also
int hp;
  double speed;

public:
  // Mammal constructor - runs FIRST before derived class ctors!
  Mammal()
  {
    hp = 100;
    speed = 1.0;
    cout << "A mammal is created!" << endl;
  }
  ~Mammal()
```

```
  {
    cout << "A mammal has fallen!" << endl;
  }
  // Common function to all Mammals and derivatives
  void breathe()
  {
    cout << "Breathe in.. breathe out" << endl;
  }
  virtual void talk()
  {
    cout << "Mammal talk.. override this function!" << endl;
  }
  // pure virtual function, (explained below)
  virtual void walk() = 0;
};

// This next line says "class Dog inherits from class Mammal"
class Dog : public Mammal // : is used for inheritance
{
public:
  Dog()
  {
    cout << "A dog is born!" << endl;
  }
  ~Dog()
  {
    cout << "The dog died" << endl;
  }
  virtual void talk() override
  {
    cout << "Woof!" << endl; // dogs only say woof!
  }
  // implements walking for a dog
  virtual void walk() override
  {
    cout << "Left front paw & back right paw, right front paw &
     back left paw.. at the speed of " << speed << endl;
  }
};

class Cat : public Mammal
{
public:
  Cat()
  {
    cout << "A cat is born" << endl;
  }
  ~Cat()
```

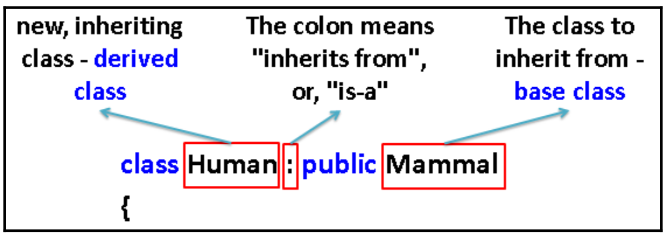
```
  {
    cout << "The cat has died" << endl;
  }
  virtual void talk() override
  {
    cout << "Meow!" << endl;
  }
  // implements walking for a cat.. same as dog!
  virtual void walk() override
  {
    cout << "Left front paw & back right paw, right front paw &
     back left paw.. at the speed of " << speed << endl;
  }
};

class Human : public Mammal
{
  // Data member unique to Human (not found in other Mammals)
  bool civilized;
public:
  Human()
  {
    cout << "A new human is born" << endl;
    speed = 2.0; // change speed. Since derived class ctor
    // (ctor is short for constructor!) runs after base
    // class ctor, initialization sticks initialize member
    // variables specific to this class
    civilized = true;
  }
  ~Human()
  {
    cout << "The human has died" << endl;
  }
  virtual void talk() override
  {
    cout << "I'm good looking for a .. human" << endl;
  }
  // implements walking for a human..
  virtual void walk() override
  {
    cout << "Left, right, left, right at the speed of " << speed
     << endl;
  }
  // member function unique to human derivative
  void attack( Human & other )
  {
    // Human refuses to attack if civilized
    if( civilized )
```

```
        cout << "Why would a human attack another? I refuse" <<
         endl;
      else
        cout << "A human attacks another!" << endl;
    }
};

int main()
{
  Human human;
  human.breathe(); // breathe using Mammal base class
   functionality
  human.talk();
  human.walk();

  Cat cat;
  cat.breathe(); // breathe using Mammal base class functionality
  cat.talk();
  cat.walk();

  Dog dog;
  dog.breathe();
  dog.talk();
  dog.walk();
}
```

All of `Dog`, `Cat`, and `Human` inherit from `class Mammal`. This means that `dog`, `cat`, and `human` are mammals, and many more.

Syntax of inheritance

The syntax of inheritance is quite simple. Let's take the `Human` class definition as an example. The following screenshot is a typical inheritance statement:



The class on the left of the colon (:) is the new, derived class, and the class on the right of the colon is the base class.

What does inheritance do?

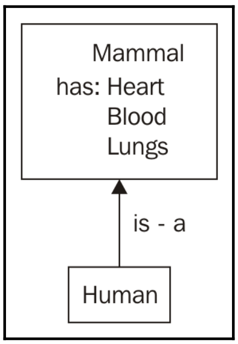
The point of inheritance is for the derived class to take on all the characteristics (data members and member functions) of the base class, and then to extend it with even more functionality. For instance, all mammals have a `breathe()` function. By inheriting from the `Mammal` class, the `Dog`, `Cat`, and `Human` classes all automatically gain the ability to `breathe()`.

Inheritance reduces replication of code, since we don't have to re-implement common functionalities (such as `.breathe()`) for `Dog`, `Cat`, and `Human`. Instead, each of these derived classes enjoys the reuse of the `breathe()` function defined in `class Mammal`.

However, only the `Human` class has the `attack()` member function. This would mean that, in our code, only the `Human` class attacks. The `cat.attack()` function will introduce a compiler error, unless you write an `attack()` member function inside `class Cat` (or in `class Mammal`).

The is-a relationship

Inheritance is often said to be an `is-a` relationship. When a `Human` class inherits from the `Mammal` class, then we say that a human *is-a* mammal:

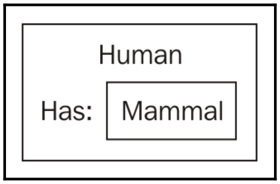


A human inherits all the traits a mammal has.

But what if a `Human` object contains a `Mammal` object inside it, as follows?

```
class Human
{
  Mammal mammal;
};
```

In this example, we would say the human has a `Mammal` on it somewhere (which would make sense if the human were pregnant, or somehow carrying a mammal):



This `Human` class instance has some kind of mammal attached to it

Remember that we previously gave `Player` an `Armor` object inside it? It wouldn't make sense for the `Player` object to inherit from the `Armor` class, because it wouldn't make sense to say the `Player` *is-an Armor*. When deciding whether one class inherits from another or not in code design (for example, the `Human` class inherits from the `Mammal` class), you must always be able to comfortably say something like the `Human` class *is-a* `Mammal`. If the *is-a* statement sounds wrong, then it is likely that inheritance is the wrong relationship for that pair of objects.

In the preceding example, we're introducing a few new C++ keywords. The first is `protected`.

Protected variables

A `protected` member variable is different from a `public` or `private` variable. All three classes of variables are accessible inside the class in which they are defined. The difference between them is with regard to accessibility outside the class. A `public` variable is accessible anywhere inside the class and outside the class. A `private` variable is accessible inside the class but not outside the class. A `protected` variable is accessible inside the class, and inside of derived subclasses, but is not accessible outside the class. So, the `hp` and `speed` members of `class Mammal` will be accessible in the derived classes `Dog`, `Cat`, `Horse`, and `Human`, but not outside of these classes (in `main()`, for instance).

Virtual functions

A virtual function is a member function whose implementation can be overridden in a derived class. In this example, the `talk()` member function (defined in `class Mammal`) is marked `virtual`. This means that the derived classes might or might not choose to implement their own version of what the `talk()` member function means.

Purely virtual functions

A purely `virtual` function (and abstract classes) is one whose implementation you are required to override in the derived class. The `walk()` function in `class Mammal` is purely virtual; it was declared like this:

```
virtual void walk() = 0;
```

The `= 0` part at the end of the preceding code is what makes the function purely virtual.

The `walk()` function in `class Mammal` is purely virtual and this makes the `Mammal` class abstract. An abstract class in C++ is any class that has at least one purely virtual function.

If a class contains a purely virtual function and is abstract, then that class cannot be instantiated directly. That is, you cannot create a `Mammal` object now, on account of the purely virtual function `walk()`. If you tried to do the following code, you would get an error:

```
int main()
{
  Mammal mammal;
}
```

If you try to create a `Mammal` object, you will get the following error:

```
error C2259: 'Mammal' : cannot instantiate abstract class
```

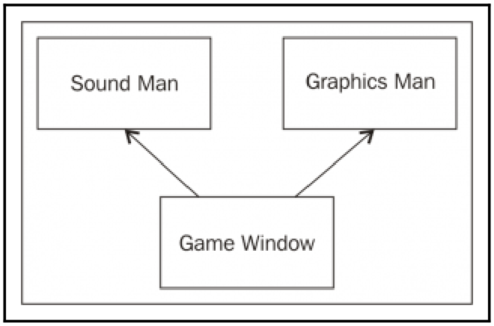
You can, however, create instances of derivatives of `class Mammal`, as long as the derived classes have all of the purely virtual member functions implemented.

You may wonder why you'd want to use one of these. Well, do you really think you'd want to create a `Mammal` object in a game? No, you'd want to create an object of the types you derive from `Mammal`, such as `Cat` or `Dog`. This way, you can't accidentally create a `Mammal`, which would be very confusing to the `Player`!

Multiple inheritance

Not everything multiple is as good as it sounds. Multiple inheritance is when a derived class inherits from more than one base class. Usually, this works without a hitch if the multiple base classes we are inheriting from are completely unrelated.

For example, we can have a class `Window` that inherits from the `SoundManager` and `GraphicsManager` base classes. If `SoundManager` provides a member function `playSound()` and `GraphicsManager` provides a member function `drawSprite()`, then the `Window` class will be able to use those additional capabilities without a hitch:



Game Window inheriting from Sound Man and Graphics Man means Game Window will have both sets of capabilities

However, multiple inheritance can have negative consequences. Say we want to create a `Mule` class that derives from both the `Donkey` and `Horse` classes. The `Donkey` and `Horse` classes, however, both inherit from the `Mammal` base class. We instantly have an issue! If we were to call `mule.talk()`, but `mule` did not override the `talk()` function, which member function should be invoked, that of `Horse` or `Donkey`? It's ambiguous.

Private inheritance

A less talked about feature of C++ is `private` inheritance. Whenever a class inherits from another class publicly, it is known to all code whose parent class it belongs to, for example:

```
class Cat : public Mammal
```

This means that all code will know that `Cat` is an object of `Mammal`, and it will be possible to point to a `Cat*` instance using a base class `Mammal*` pointer. For example, the following code would be valid:

```
Cat cat;
Mammal* mammalPtr = &cat; // Point to the Cat as if it were a
                          // Mammal
```

Putting an object of one class into a variable of the type of the parent class is called casting. The preceding code is fine if `Cat` inherits from `Mammal` publicly. Private inheritance is where code outside the `Cat` class is not allowed to know the parent class:

```
class Cat : private Mammal
```

Here, externally called code will not "know" that the `Cat` class derives from the `Mammal` class. Casting a `Cat` instance to the `Mammal` base class is not allowed by the compiler when inheritance is `private`. Use `private` inheritance when you need to hide the fact that a certain class derives from a certain parent class.

However, `private` inheritance is rarely used in practice. Most classes just use `public` inheritance. If you want to know more about `private` inheritance, see `http://stackoverflow.com/questions/406081/why-should-i-avoid-multiple-inheritance-in-c`.

Putting your classes into headers

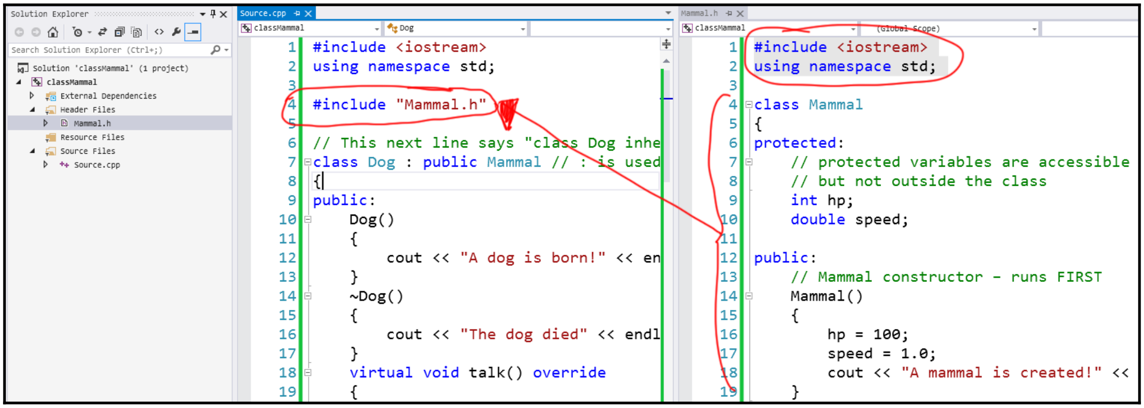
So far, our classes have just been pasted before `main()`. If you continue to program that way, your code will all be in one file and will appear as one big disorganized mess.

Therefore, it is a good programming practice to organize your classes into separate files. This makes editing each class's code individually much easier when there are multiple classes inside the project.

Take `class Mammal` and its derived classes from earlier. We will properly organize that example into separate files. Let's do this in steps:

1. Create a new file in your C++ project called `Mammal.h`. Cut and paste the entire `Mammal` class into that file. Notice that, since the `Mammal` class included the use of `cout`, we write a `#include <iostream>` statement in that file as well.
2. Write an `"#includeMammal.h"` statement at the top of your `Source.cpp` file.

An example of what this looks like is shown in the following screenshot:



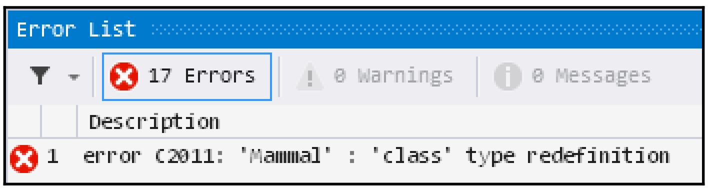
What's happening here when the code is compiled is that the entire `Mammal` class is copied and pasted (`#include`) into the `Source.cpp` file, which contains the `main()` function, and the rest of the classes are derived from `Mammal`. Since `#include` is a copy and paste function, the code will function exactly the same as it did before; the only difference is that it will be much better organized and easier to look at. Compile and run your code during this step to make sure it still works.

> **TIP**
> Check that your code compiles and runs often, especially when refactoring. When you don't know the rules, you're bound to make a lot of mistakes. This is why you should do your refactoring only in small steps. Refactoring is the name for the activity we are doing now - we are reorganizing the source to make more sense to other readers of our code base. Refactoring usually does not involve rewriting too much of it.

The next thing you need to do is isolate the `Dog`, `Cat`, and `Human` classes into their own files. To do so, create the `Dog.h`, `Cat.h`, and `Human.h` files and add them to your project.

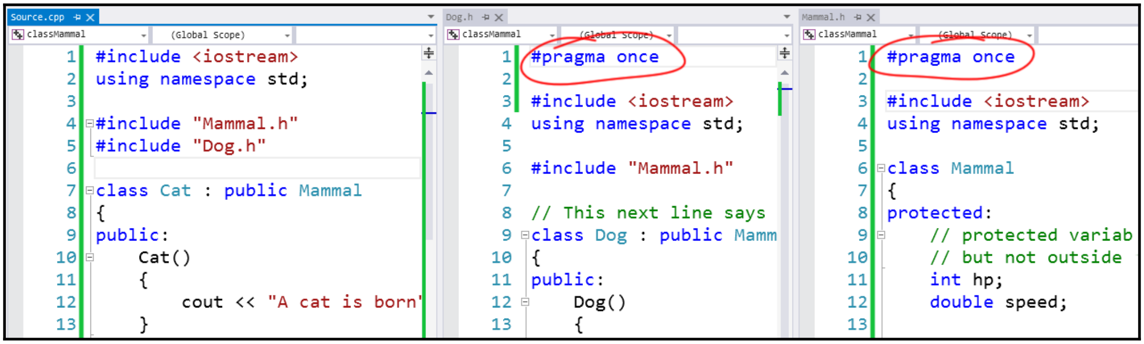
Let's start with the `Dog` class, as shown in the following screenshot.

If you use exactly this setup and try to compile and run your project, you will see the **'Mammal' : 'class' type redefinition** error, as shown in the following screenshot:



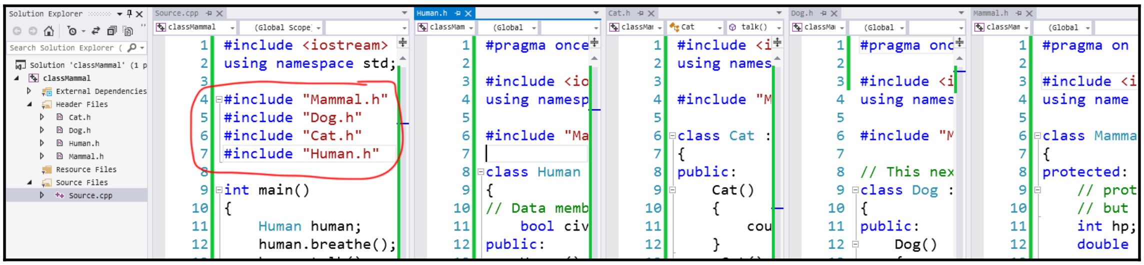
What this error means is that `Mammal.h` has been included twice in your project, once in `Source.cpp` and then again in `Dog.h`. This means, effectively, two versions of the `Mammal` class got added to the compiling code, and C++ is unsure which version to use.

There are a few ways to fix this issue, but the easiest (and the one that Unreal Engine uses) is the `#pragma once` macro, as shown in the following screenshot:



We write `#pragma once` at the top of each header file. This way, the second time `Mammal.h` is included, the compiler doesn't copy and paste its contents again, since it has already been included before, and its content is actually already in the compiling group of files.

Do the same thing for `Cat.h` and `Human.h`, then `include` them both in your `Source.cpp` file where your `main()` function resides:



Screenshot with all classes included

Now that we've included all classes into your project, the code should compile and run.

Using .h and .cpp files

The next level of organization is to leave the class declarations in the header files (`.h`) and put the actual function implementation bodies inside some new `.cpp` files. Also, leave existing members inside the `class Mammal` declaration.

For each class, perform the following operations:

1. Delete all function bodies (code between `{` and `}`) and replace them with just a semicolon. For the `Mammal` class, this would look as follows:

```
// Mammal.h
#pragma once
class Mammal
{
protected:
  int hp;
  double speed;

public:
  Mammal();
  ~Mammal();
  void breathe();
  virtual void talk();
  // pure virtual function,
  virtual void walk() = 0;
};
```

2. Create a new `.cpp` file called `Mammal.cpp`. Then, simply put the member function bodies inside this file:

```
// Mammal.cpp
#include <iostream>
using namespace std;

#include "Mammal.h"
Mammal::Mammal() // Notice use of :: (scope resolution
operator)
{
  hp = 100;
  speed = 1.0;
  cout << "A mammal is created!" << endl;
}
Mammal::~Mammal()
{
  cout << "A mammal has fallen!" << endl;
}
```

```
void Mammal::breathe()
{
  cout << "Breathe in.. breathe out" << endl;
}
void Mammal::talk()
{
  cout << "Mammal talk.. override this function!" << endl;
}
```

It is important to note the use of the class name and scope resolution operator (double colon) when declaring the member function bodies. We prefix all member functions belonging to the `Mammal` class with `Mammal::`. This shows that they belong to the class (this differentiates them from `.`, which is used for specific object instances of that class type).

Notice how the purely virtual function does not have a body; it's not supposed to! Purely virtual functions are simply declared (and initialized to `0`) in the base class, but implemented later in derived classes.

Exercise

Complete the separation of the different creature classes above into class header (`.h`) and class definition files (`.cpp`).

Object-oriented programming design patterns

If you've been looking into programming, you have probably come across the term *design patterns*. Design patterns are important to know because they are standard ways of doing things that can be applied to many programming projects. For an in-depth look at design patterns if you want to know more, a classic book is *Design Patterns* (`https://www.goodreads.com/book/show/85009.Design_Patterns`). Once you familiar them, you will find many uses for them throughout your career. Not all specifically relate to objects, but here are a few examples that do.

Singletons

Sometimes, you only want to have one instance of an object. Say you're doing a kingdom simulator. You only want there to be one king. Otherwise, you risk a *Game of Thrones*-type situation with intrigue and Red Weddings everywhere, and that's not the type of game you were aiming for, is it? (Of course, you might keep that in mind for a different game.) But for this particular game, you want one king running things.

So, how can you make sure other kings don't start turning up everywhere? You use a singleton. A singleton is a class that keeps an instance of an object, and anywhere you want to use it, instead of creating a new object, you call a function that gives you a way to access an instance of an object, and you can then call functions on that. To make sure you only create one instance of an object, it keeps a copy of itself in a static variable inside the class (note: we will be talking more about static class members in the next section), and when you call `GetInstance()`, it checks to see whether you've already created an instance of the object. If you have, it uses the existing one. If you haven't, it creates a new one. Here's an example:

```
//King.h

#pragma once
#include <string>

using namespace std;

class King
{
public:
    ~King();

    static King* getInstance();

    void setName(string n) { name = n; };
    string getName() const { return name; };
    //Add more functions for King
private:
    King();

    static King* instance;
    string name;
};
```

Here is the code for the `cpp`:

```
//King.cpp

#include "King.h"

King* King::instance = nullptr;

King::King()
{
}

King::~King()
{
}

King* King::getInstance()
{
    if (instance == nullptr)
    {
        instance = new King();
    }
    return instance;
}
```

The constructor is listed in the `private:` section of the code. This is important. If you do this, the constructor will not be accessible from outside the class, meaning that no other programmers, who may not realize that this is a singleton, can start creating new `King` objects and wreak havoc on the game. If they try, they will get an error. So, this enforces that this class can only be accessed through the `getInstance()` function.

To use this new singleton class, you would do something like this:

```
King::getInstance()->setName("Arthur");
cout << "I am King " << King::getInstance()->getName();
```

Once you set the name, it will output `I am King Arthur`, no matter where in the code you call it from (just make sure to add `#include "King.h"` at the top of the file).

Factories

What do you think of when you think of the term *factory*? Probably a place where they mass-produce objects, such as cars, shoes, or computers. In code, a `Factory` works the same way. A factory is a class that can create objects of other types. But it's even more flexible because it can create objects of different types.

We saw earlier that a mammal can be a dog, cat, horse, or human. Because all four types are derived from `Mammal`, a `Factory` object can have a function where you tell it which type of `Mammal` you want, and it will create an object of that type, do any setup necessary, and return it. Because of a principle called polymorphism, you can get an object of type `Mammal`, but when you call any virtual functions, it knows to use the ones for `Cat`, `Dog`, or `Human`, depending on the type of object created. Your C++ compiler knows this because it maintains a virtual function table behind the scenes, which keeps a pointer to the version of each virtual function you really want to be using, and stores those in each object.

Object pools

Say you're creating a lot of objects, such as a particle system to display fireworks, and you have to constantly create new firework animations all over the screen. After a while, you'll notice things slowing down, and you might even run out of memory and crash. Fortunately, there is a way around this.

You can create an object pool, which is basically a group of objects that should be large enough to contain every one on screen at any given time. When one finishes its animation and disappears, instead of creating a new one, you throw it back into the pool, and when you need another one, you can pull that one back out and reuse it (you may want to change the color or other settings first). Reusing objects from a pool is much faster and takes less processing than creating new objects constantly. It also helps avoid memory leaks.

Static members

As we saw in the singleton example, classes can have static members. A static member of a class exists once for all instances of the class, instead of being different for each one. You generally access them as we did for the singleton:

```
King::getInstance()->setName("Arthur");
```

Static variables are also commonly used for constants related to a class. But they can also be used to track something, such as how many instances of an object you have, by incrementing the static variable in a constructor and then decrementing it in the destructor. This is similar to how smart pointers can keep track of how many references to an object still exist.

Callable objects and invoke

Another new C++ feature is callable objects. This is an advanced topic, so don't worry too much about understanding it at this point, but I'll give you a brief overview. But to explain it, first, I need to mention another topic — operator overloading.

You may think you can't change the meaning of operators such as `+`, `-`, `*`, and `/`. Actually, in C++, you can. You can add a function called `operator(symbol)`. So, if you have a string class, you can create an `operator+` function that causes strings to be concatenated instead of trying to figure out how to add two objects that aren't actually numbers.

Callable objects go even further by overriding `()` with `operator()`. So, you can have a class that can be called as an object. C++ 17 has added a new function, `invoke()`, that will let you call a callable object with parameters.

Summary

In this chapter, you learned about objects in C++; they are pieces of code that tie data members and member functions together into a bundle of code called a `class` or `struct`. Object-oriented programming means that your code will be filled with things instead of just `int`, `float`, and `char` variables. You will have a variable that represents `Barrel`, another variable that represents `Player`, and so on, that is, a variable to represent every entity in your game. You will be able to reuse code by using inheritance; if you have to code implementations of `Cat` and `Dog`, you can code a common functionality in the base class `Mammal`. We also discussed encapsulation and how it is easier and more efficient to program objects such that they maintain their own internal state. We also introduced a few design patterns for objects (you'll find there are many more out there).

In the next chapter, we'll talk about how to allocate memory dynamically, and about arrays and vectors.

7 Dynamic Memory Allocation

In the previous chapter, we talked about class definitions and how to devise your own custom class. We discussed how, by devising your own custom classes, you can construct variables that represent entities within your game or program.

In this chapter, we will talk about dynamic memory allocation and how to create space in memory for groups of objects. Let's take a look at the topics covered in this chapter:

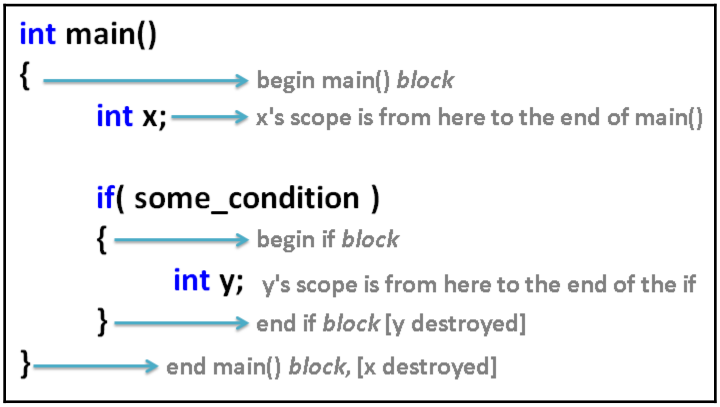
- Constructors and destructors revisited
- Dynamic memory allocation
- Regular arrays
- C++ style dynamic size arrays (new[] and delete[])
- Dynamic C-style arrays
- Vectors

Constructors and destructors revisited

Assume that we have a simplified version of `class Player`, as before, with only a constructor and a destructor:

```
class Player
{
  string name;
  int hp;
public:
  Player(){ cout << "Player born" << endl; }
  ~Player(){ cout << "Player died" << endl; }
};
```

We talked earlier about the *scope* of a variable in C++; to recap, the scope of a variable is the section of the program where that variable can be used. The scope of a variable is generally inside the block in which it was declared. A block is just any section of code contained between `{` and `}`. Here is a sample program that illustrates variable scope:



In this sample program, the x variable has scope through all of main(). The y variable's scope is only inside the if block.

We mentioned previously that, in general, variables are destroyed when they go out of scope. Let's test this idea out with instances of `class Player`:

```
int main()
{
  Player player; // "Player born"
}                // "Player died" - player object destroyed here
```

The output of this program is as follows:

```
Player born
Player died
```

The destructor for the `Player` object is called at the end of the player object's scope. Since the scope of a variable is the block within which it is defined in the three lines of code, the `Player` object would be destroyed immediately at the end of `main()`, when it goes out of scope.

Dynamic memory allocation

Now, let's try allocating a `Player` object dynamically. What does that mean?

We use the `new` keyword to allocate it:

```
int main()
{
  // "dynamic allocation" - using keyword new!
  // this style of allocation means that the player object will
  // NOT be deleted automatically at the end of the block where
  // it was declared! Note: new always returns a pointer
Player *player = new Player();
} // NO automatic deletion!
```

The output of this program is as follows:

```
Player born
```

The player does not die! How do we kill the player? We must explicitly call `delete` on the `player` pointer.

The delete keyword

The `delete` operator invokes the destructor on the object being deleted, as shown in the following code:

```
int main()
{
  // "dynamic allocation" - using keyword new!
  Player *player = new Player();
  delete player; // deletion invokes dtor
}
```

The output of the program is as follows:

```
Player born
Player died
```

So, only normal (or automatic, also called non-pointer type) variable types get destroyed at the end of the block in which they were declared. Pointer types (variables declared with `*` and `new`) are not automatically destroyed, even when they go out of scope.

What is the use of this? Dynamic allocation lets you control when an object is created and destroyed. This will come in handy later.

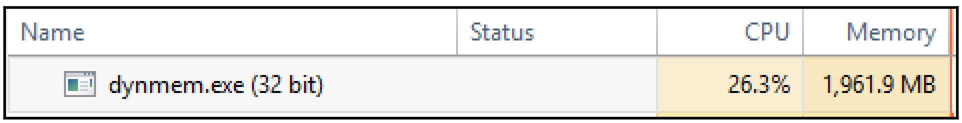
Memory leaks

So, dynamically allocated objects created with `new` are not automatically deleted, unless you explicitly call `delete` on them. There is a risk here! It is called a *memory leak*. Memory leaks happen when an object allocated with `new` is not ever deleted. What can happen is that, if a lot of objects in your program are allocated with `new` and then you stop using them, your computer will eventually run out of memory due to memory leakage.

Here is a ridiculous sample program to illustrate the problem:

```
#include <iostream>
#include <string>
using namespace std;
class Player
{
  string name;
  int hp;
public:
  Player(){ cout << "Player born" << endl; }
  ~Player(){ cout << "Player died" << endl; }
};

int main()
{
  while( true ) // keep going forever,
  {
    // alloc..
    Player *player = new Player();
    // without delete == Memory Leak!
  }
}
```

This program, if left to run long enough, will eventually gobble the computer's memory, as shown in the following screenshot:



2 GB of RAM used for Player objects.

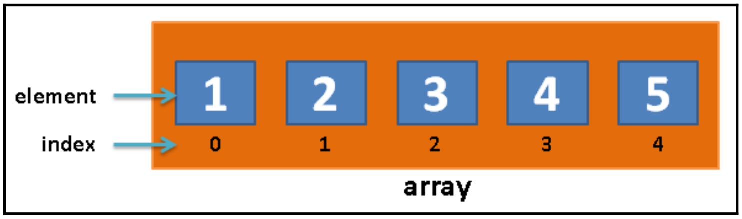
Note that no one ever intends to write a program with this type of problem in it! Memory leak problems happen accidentally. You must take care of your memory allocation and `delete` objects that are no longer in use.

Regular arrays

An array in C++ can be declared as follows:

```
#include <iostream>
using namespace std;
int main()
{
  int array[ 5 ];  // declare an "array" of 5 integers
                   // fill slots 0-4 with values
array[ 0 ] = 1;
array[ 1 ] = 2;
array[ 2 ] = 3;
array[ 3 ] = 4;
array[ 4 ] = 5;
  // print out the contents
  for( int index = 0; index < 5; index++ )
    cout << array[ index ] << endl;
}
```

The way this looks in memory is something like this:



That is, inside the `array` variable are five slots or elements. Inside each of the slots is a regular `int` variable. You can also declare an array by passing in values, like this:

```
int array[ ] = {6, 0, 5, 19};
```

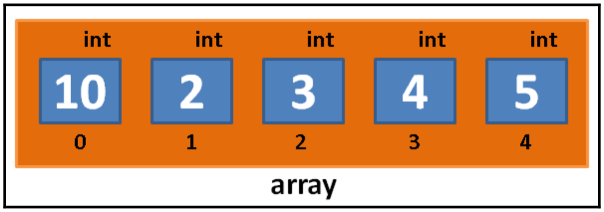
You can also pass in `int` variables to use the values stored there.

The array syntax

So, how do you access one of the `int` values in the array? To access the individual elements of an array, we use square brackets, as shown in the following line of code:

```
array[ 0 ] = 10;
```

It's very similar to the syntax for creating an array in the first place. The preceding line of code would change the element at slot `0` of the array to a `10`:



In general, to get to a particular slot of an array, you will write the following:

```
array[ slotNumber ] = value to put into array;
```

Keep in mind that array slots are always indexed starting from `0` (some languages may start from `1`, but that's unusual and could get confusing). To get into the first slot of the array, use `array[0]`. The second slot of the array is `array[1]` (not `array[2]`). The final slot of the preceding array is `array[4]` (not `array[5]`). The `array[5]` data type is out of bounds of the array! (There is no slot with index 5 in the preceding diagram. The highest index is 4.)

Don't go out of the bounds of the array! It might work sometimes, but other times your program will crash with a **memory access violation** (accessing memory that doesn't belong to your program). In general, accessing memory that does not belong to your program will will cause your app to crash, and if it doesn't do so immediately, there will be a hidden bug in your program that will only cause problems once in a while. You must always be careful when indexing an array.

Arrays are built into C++, that is, you don't need to include anything special to have immediate use of arrays. You can have arrays of any type of data that you want, for example, arrays of `int`, `double`, `string`, and even your own custom object types (`Player`).

Exercise

1. Create an array of five strings and put some names inside it(made up or random—it doesn't matter).
2. Create an array of doubles called `temps` with three elements and store the temperature for the last three days in it.

Solution

1. The following is a sample program with an array of five strings:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string array[ 5 ];  // declare an "array" of 5 strings
                      // fill slots 0-4 with values
array[ 0 ] = "Mariam McGonical";
array[ 1 ] = "Wesley Snice";
array[ 2 ] = "Kate Winslett";
array[ 3 ] = "Erika Badu";
array[ 4 ] = "Mohammad";
  // print out the contents
  for( int index = 0; index < 5; index++ )
    cout << array[ index ] << endl;
}
```

2. The following is just the array:

```
double temps[ 3 ];
// fill slots 0-2 with values
temps[ 0 ] = 0;
temps[ 1 ] = 4.5;
temps[ 2 ] = 11;
```

C++ style dynamic size arrays (new[] and delete[])

It probably occurred to you that we won't always know the size of an array at the start of a program. We would need to allocate the array's size dynamically.

However, if you've tried it, you might have noticed that this doesn't work!

Let's try and use the `cin` command to take in an array size from the user. Let's ask the user how big he wants his array and try to create one of that size for him:

```
#include <iostream>
using namespace std;
int main()
{
```

```
    cout << "How big?" << endl;
    int size;       // try and use a variable for size..
    cin >> size;    // get size from user
    int array[ size ];  // get error
  }
```

We get an error. The problem is that the compiler wants to allocate the size of the array. However, unless the variable size is marked `const`, the compiler will not be sure of its value at compile time. The C++ compiler cannot size the array at compile time, so it generates a compile time error.

To fix this, we have to allocate the array dynamically (on the "heap"):

```
  #include <iostream>
  using namespace std;
  int main()
  {
    cout << "How big?" << endl;
    int size;       // try and use a variable for size..
    cin >> size;
    int *array = new int[ size ];  // this works
    // fill the array and print
  for( int index = 0; index < size; index++ )
  {
    array[ index ] = index * 2;
    cout << array[ index ] << endl;
  }
  delete[] array; // must call delete[] on array allocated with
                  // new[]!
  }
```

So, the lessons here are as follows:

- To allocate an array of some type (for example, `int`) dynamically, you must use new `int[numberOfElementsInArray]`.
- Arrays allocated with `new[]` must be deleted later with `delete[]`, otherwise, you'll get a memory leak (that's `delete[]` with square brackets; not regular delete)!

Dynamic C-style arrays

C-style arrays are a legacy topic, but they are still worth discussing, since even though they are old, you might still see them used sometimes.

The way we declare a C-style array is as follows:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "How big?" << endl;
  int size;       // try and use a variable for size..
  cin >> size;
  // the next line will look weird..
  int *array = (int*)malloc( size*sizeof(int) ); // C-style
  // fill the array and print
for( int index = 0; index < size; index++ )
  {
    //At this point the syntax is the same as with regular arrays.
    array[ index ] = index * 2;
    cout << array[ index ] << endl;
  }
free( array ); // must call free() on array allocated with
               // malloc() (not delete[]!)
}
```

The differences are highlighted here.

A C-style array is created using the `malloc()` function. The word malloc stands for memory allocate. This function requires you to pass in the size of the array in bytes to create, and not just the number of elements you want in the array. For this reason, we multiply the number of elements requested (`size`) by `sizeof` of the type inside the array. The sizes in bytes of a few typical C++ types are listed in the following table:

C++ primitive type	`sizeof` (size in bytes)
`int`	4
`float`	4
`double`	8
`long long`	8

Memory allocated with the `malloc()` function must later be released using `free()`.

Vectors

There's one other way of creating what are essentially arrays, and this one is the easiest to use and is preferred by many programmers—using a vector. Imagine that, in any of the previous examples, you were adding new items into an array and suddenly ran out of space while the program was running. What would you do? You could create a whole new array and copy everything over, but as you might guess, that's a lot of extra work and processing. So, what if you had a type of array that handled cases like that for you behind the scenes, without you even asking?

This is what vectors do. A vector is a member of the Standard Template Library (we will get to templates in a couple of chapters, so just be patient), and as in other examples that have come up, you can set the type inside angled brackets (<>). You create a vector like this:

```
vector<string> names; // make sure to add #include <vector> at the top
```

This basically says you are creating a vector of strings, called names. To add new items to a vector, you can use the `push_back()` function, like this:

```
names.push_back("John Smith");
```

This will add the item you pass in to the end of the vector. You can call `push_back()` as many times as you want, and whenever the vector runs out of space, it will automatically increase its size without you having to do anything! So, you can keep adding as many items as you want (within reason, since you could run out of memory eventually), without worrying about how the memory is managed.

Vectors also add other useful functions, such as `size()`, which tells you how many items a vector contains (in a standard array, you have to keep track of this yourself).

Once you have created a vector, you can treat it just like an array to access it with standard `[]` syntax:

```
//Make it unsigned int to avoid a signed/unsigned mismatch error
for (unsigned int i = 0; i < names.size(); i++)
{
    //If you get an error about << add #include <string> at the top
    cout << names[i] << endl; //endl tells it to go to the next line
}
```

Summary

This chapter introduced you to C and C++ style arrays and vectors. In most UE4 code, you will use the UE4 editor built-in collection classes (`TArray<T>`), which are similar to vectors. However, you need familiarity with basic C and C++ style arrays to be a very good C++ programmer.

We've now covered enough basic C++ to move on to UE4 next, with actors and pawns.

8
Actors and Pawns

Now, we will really delve into UE4 code. At first, it is going to look daunting. The UE4 class framework is massive, but don't worry: the framework is massive so your code doesn't have to be. You will find that you can get a lot done and a lot onto the screen using a lot less code. This is because the UE4 engine code is so extensive and well-programmed that they have made it possible to accomplish almost any game-related task easily. Just call the right functions and voila, what you want to see will appear on the screen. The entire notion of a framework is that it is designed to let you get the gameplay you want, without having to spend a lot of time sweating out the details.

The learning outcomes from this chapters are as follows:

- Actors versus pawns
- Creating a world to put your actors in
- The UE4 editor
- Starting from scratch
- Adding an actor to the scene
- Creating a player entity
- Writing C++ code that controls the game's character
- Creating non-player character entities
- Displaying a quote from each NPC dialog box

Actors versus pawns

In this chapter, we will discuss actors and pawns. Although it sounds as if pawns will be a more basic class than actors, it is actually the other way around. A UE4 actor (the `Actor` class) object is the basic type of the things that can be placed in the UE4 game world. In order to place anything in the UE4 world, you must derive from the `Actor` class.

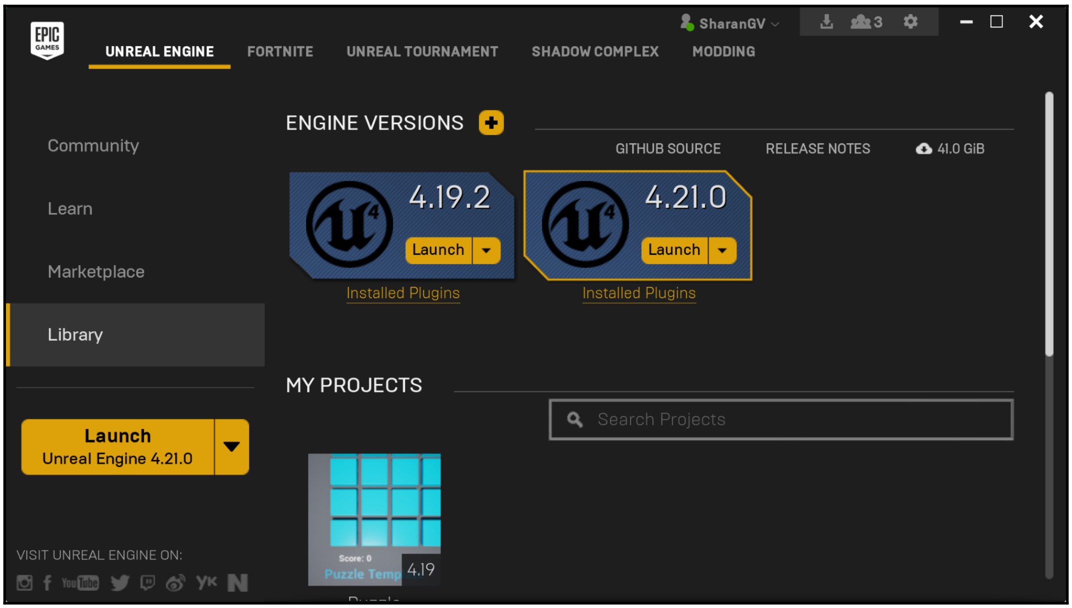
A `Pawn` is an object that represents something that you or the computer's **Artificial Intelligence** (**AI**) can control on the screen. The `Pawn` class derives from the `Actor` class, with the additional ability to be controlled by the player directly or by an AI script. When a pawn or actor is controlled by a controller or AI, it is said to be possessed by that controller or AI.

Think of the `Actor` class as a character in a play (although it could also be a prop in a play). Your game world is going to be composed of a bunch of *actors*, all acting together to make the gameplay work. The game characters, **Non-Player Characters** (**NPCs**), and even treasure chests will be actors.

Creating a world to put your actors in

Here, we will start from scratch and create a basic level into which we can put our game characters. The UE4 team has already done a great job of presenting how the world editor can be used to create a world in UE4. I want you to take a moment to create your own world by performing the following steps:

1. Create a new, blank UE4 project to get started. To do this, in the Unreal Launcher, click on the **Launch** button beside your most recent engine installation, as shown in the following screenshot:

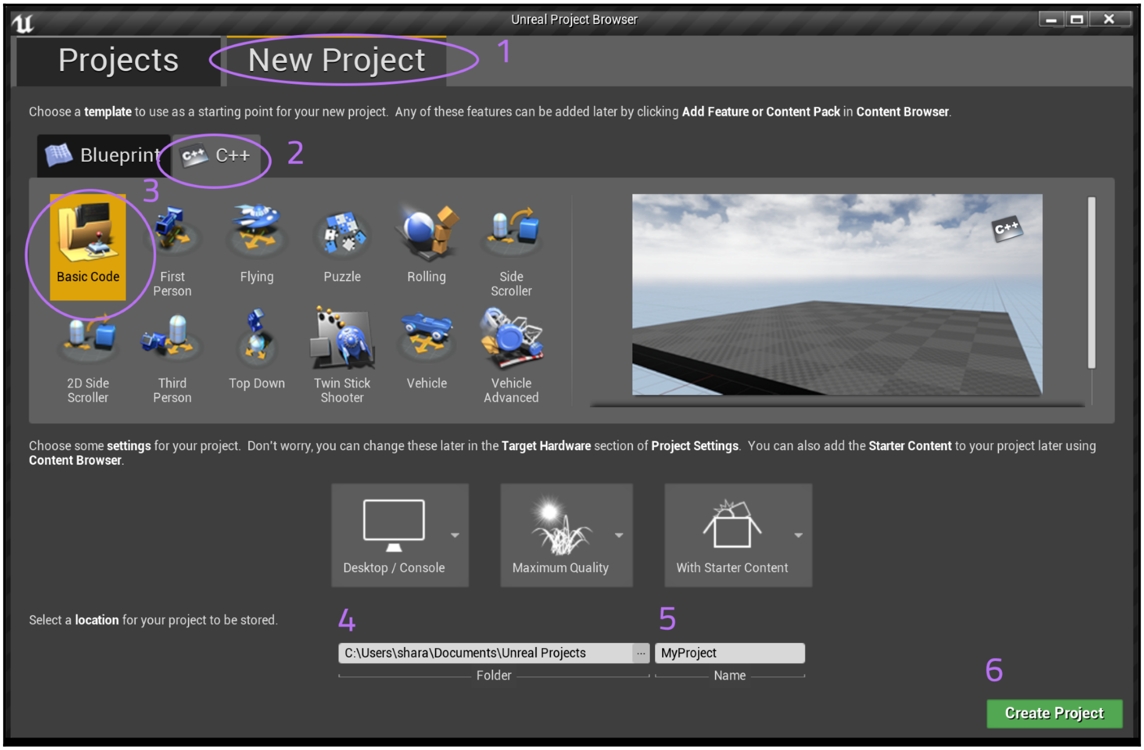


That will launch the Unreal Editor. The Unreal Editor is used to visually edit your game world. You're going to spend a lot of time in the Unreal Editor, so please take some time to experiment and play around with it.

I will only cover the basics of how to work with the UE4 editor. You will need to let your creative juices flow, however, and invest some time in order to become familiar with the editor.

> TIP
> To learn more about the UE4 editor, take a look at the *Getting Started: Introduction to the UE4 Editor* playlist, which is available at `https://www.youtube.com/playlist?list=PLZlv_N0_O1gasd4IcOe9Cx9wHoBB7rxFl`.

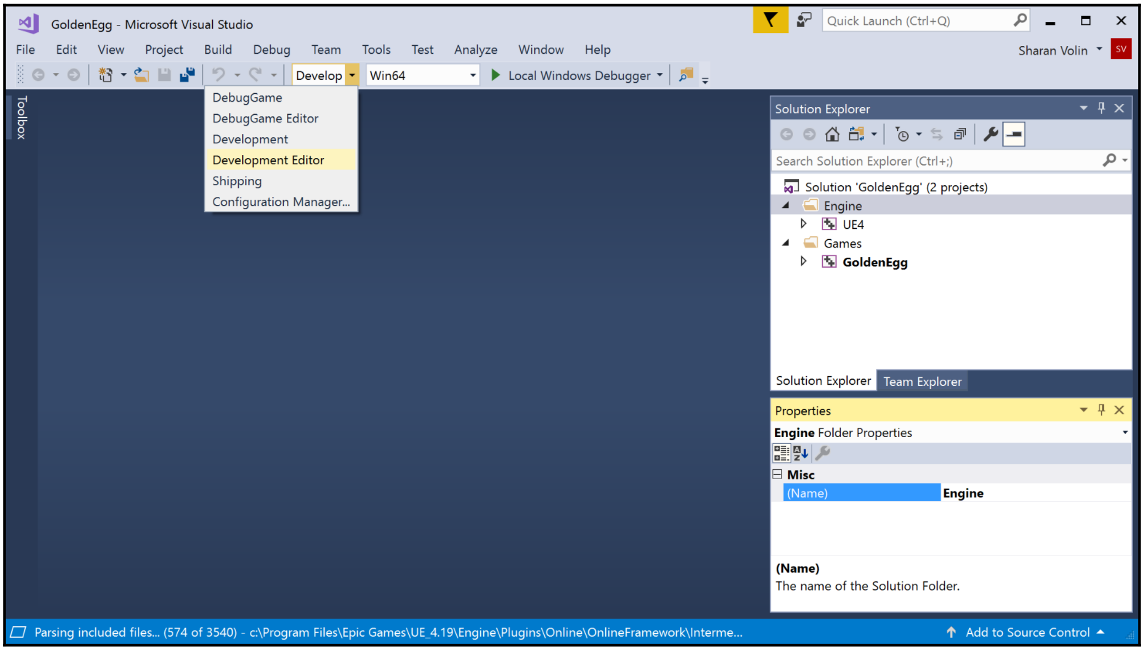
2. You will be presented with the **Projects** dialog. The following screenshot shows the steps to be performed with numbers corresponding to the order in which they need to be performed:



3. Perform the following steps to create a project:
    1. Select the **New Project** tab at the top of the screen.
    2. Click on the **C++** tab (the second sub-tab).
    3. Select **Basic Code** from the available projects listing.
    4. Set the directory where your project is located (mine is **Y:Unreal Projects**). Choose a hard disk location with a lot of space (the final project will be around 1.5 GB).
    5. Name your project. I called mine **GoldenEgg**.
    6. Click on **Create Project** to finalize project creation.

    Once you've done this, the UE4 launcher will launch Visual Studio (or Xcode). This could take a while, and the progress bar could wind up behind other windows. There will only be a couple of source files available, but we're not going to touch those now.

4. Make sure that **Development Editor** is selected from the **Configuration Manager** dropdown at the top of the screen, as shown in the following screenshot:



The Unreal Editor will also have been launched, as shown in the following screenshot:

The UE4 editor

We will explore the UE4 editor here. We'll start with the controls since it is important to know how to navigate in Unreal.

Editor controls

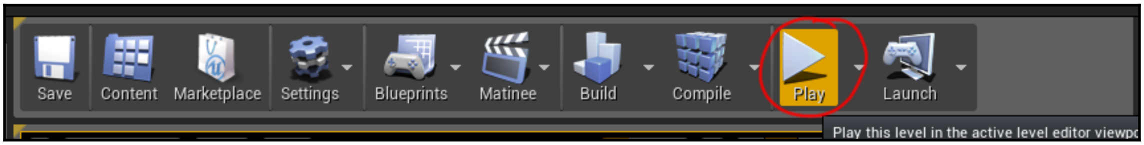
If you've never used a 3D editor before, the controls can be quite hard to learn. These are the basic navigation controls while in edit mode:

- Use the arrow keys to move around in the scene
- Press *Page Up* or *Page Down* to go up and down vertically
- Left mouse click + drag it left or right to change the direction you are facing
- Left mouse click + drag it up or down to *dolly* (move the camera forward and backward, same as pressing up/down arrow keys)

- Right mouse click + drag to change the direction you are facing
- Middle mouse click + drag to pan the view
- Right mouse click and the *W*, *A*, *S*, and *D* keys to move around the scene

Play mode controls

Click on the **Play** button in the bar at the top, as shown in the following screenshot. This will launch play mode:



Once you click on the **Play** button, the controls change. In play mode, the controls are as follows:

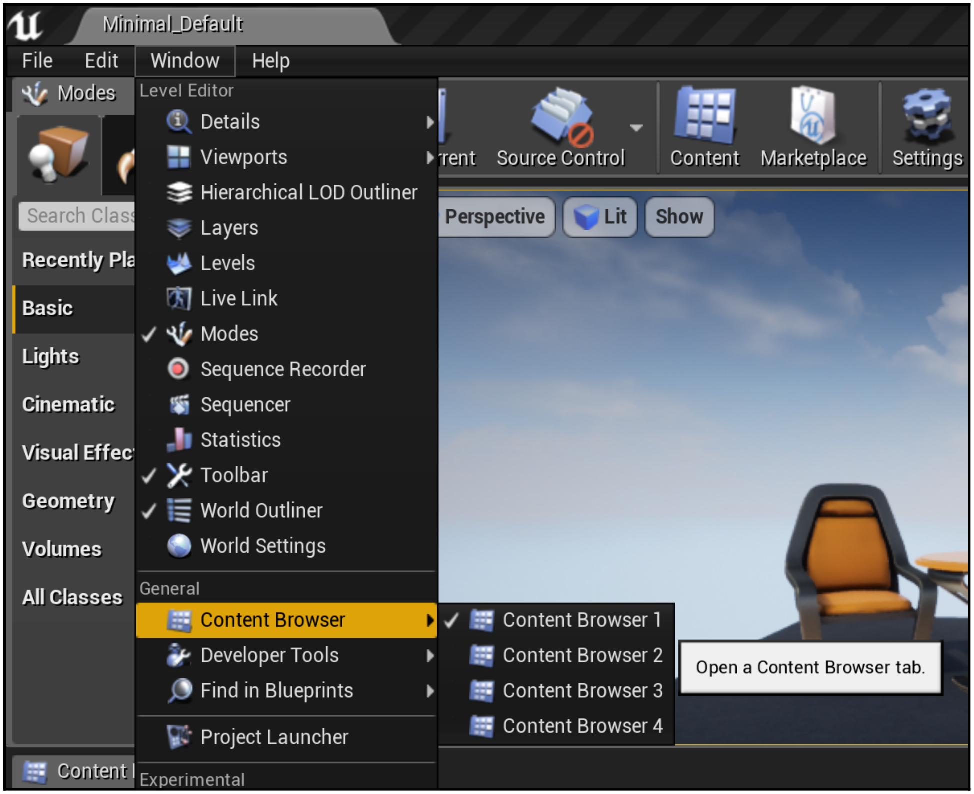
- The *W*, *A*, *S*, and *D* keys for movement
- The left or right arrow keys to look toward the left or right, respectively
- The mouse's motion to change the direction in which you look
- The *Esc* key to exit play mode and return to edit mode

At this point, I suggest you try to add a bunch of shapes and objects to the scene and try to color them with different *materials*.

Adding objects to the scene

Adding objects to the scene is as easy as dragging and dropping them in from the **Content Browser** tab, as follows:

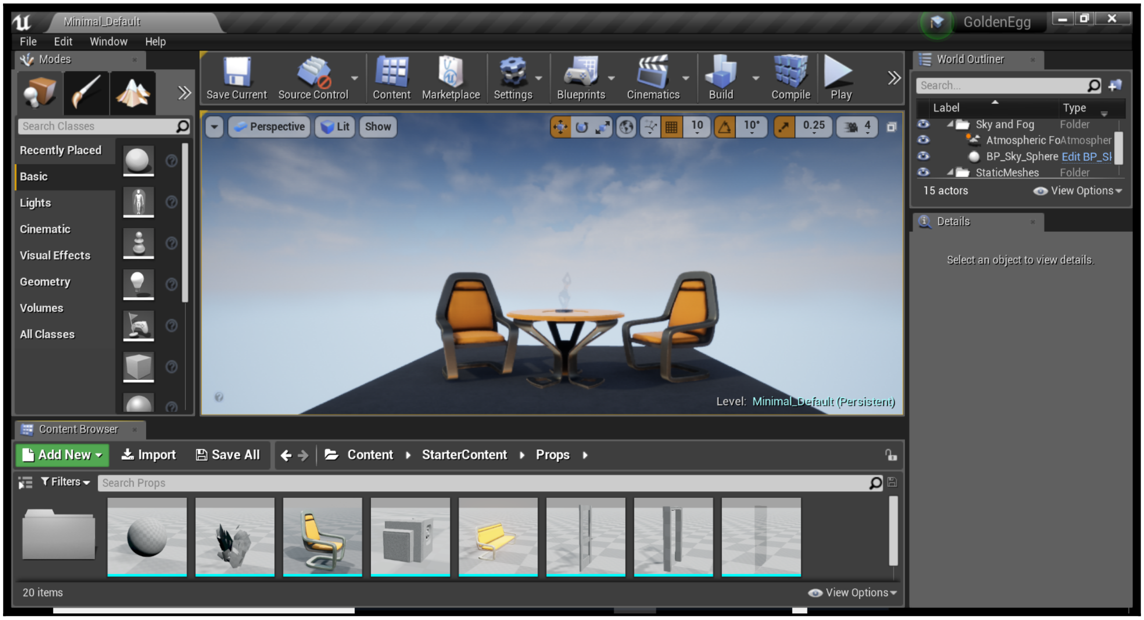
1. The **Content Browser** tab appears, by default, docked at the bottom of the window. If it isn't seen, simply select **Window** and navigate to **Content Browser** in order to make it appear:



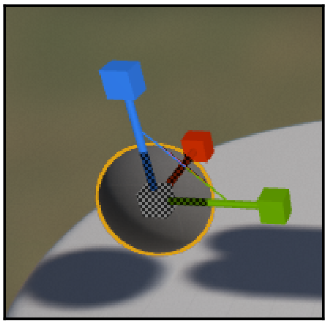
Make sure that the Content Browser is visible in order to add objects to your level

2. Double-click on the `StarterContent` folder to open it.
3. Double-click the `Props` folder to find objects you can drag into your scene.

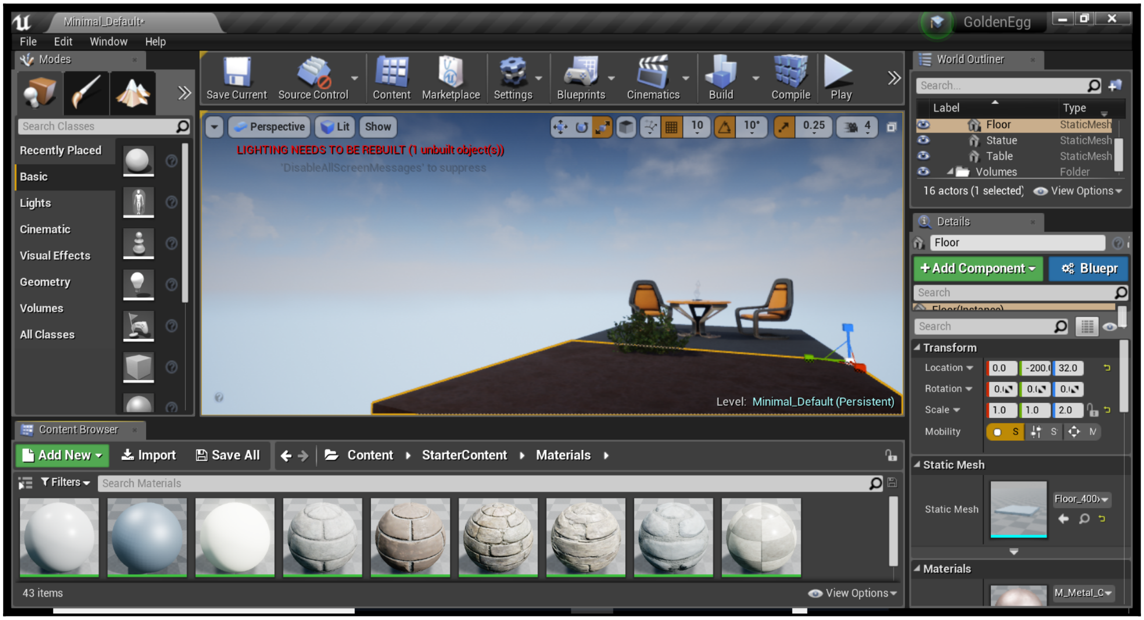
4. Drag and drop things from the **Content Browser** into your game world:



5. To resize an object, press *R* on your keyboard (hit *W* to move it again, or *E* to rotate the object). The manipulators around the object will appear as boxes, which denotes resize mode:



6. To change the material that is used to paint the object, simply drag and drop a new material from the **Content Browser** window inside the **Materials** folder:

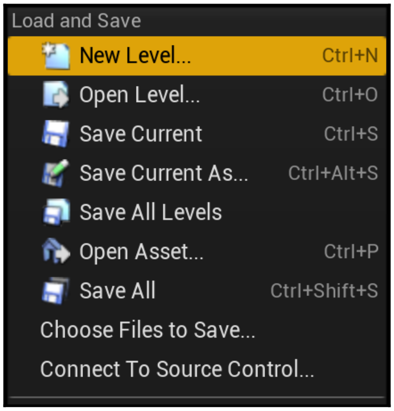


Materials are like paints. You can coat an object with any material you want by simply dragging and dropping the material you desire onto the object you desire it to be painted on. Materials are only skin deep; they don't change the other properties of an object (such as weight).

Starting a new level

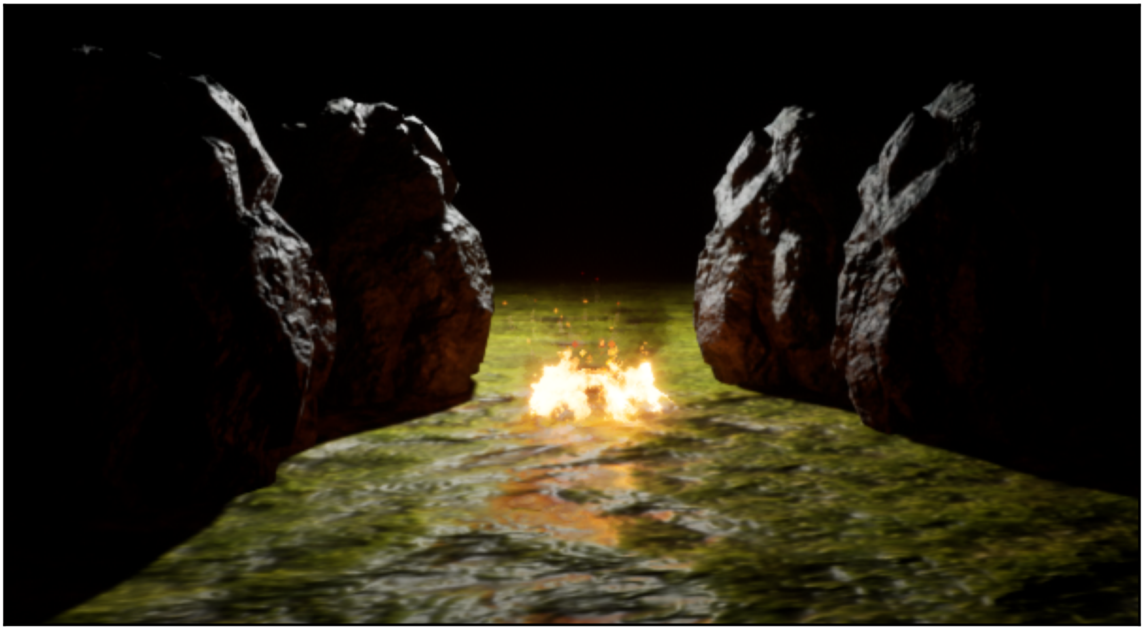
If you want to start creating a level from scratch, perform the following steps:

1. Click on **File** and navigate to **New Level...**, as shown here:

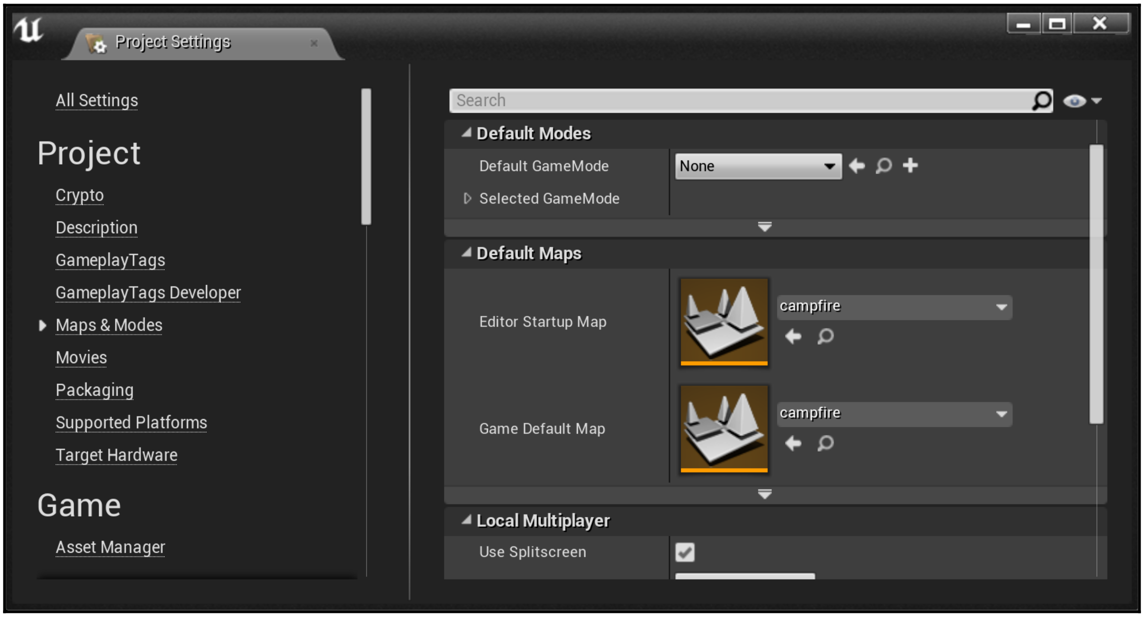


2. You can then select between **Default**, **VR-Basic**, and **Empty Level**. I think selecting **Empty Level** is a good idea:

3. The new level will be completely black to start with. Try dragging and dropping some objects from the **Content Browser** tab again.
   This time, I added a resized shapes/shape_plane for the ground plane (don't use the regular plane under modes or you'll fall through it once you add the player) and textured it with **T_ground_Moss_D**, a couple of **Props** / **SM_Rocks**, and **Particles** / **P_Fire**.
   Be sure to save your map. Here's a snapshot of my map (how does yours look?):



4. If you want to change the default level that opens when you launch the editor, go to **Edit** | **Project Settings** | **Maps & Modes**; then, you will see a **Game Default Map** and **Editor Startup Map** setting, as shown in the following screenshot:



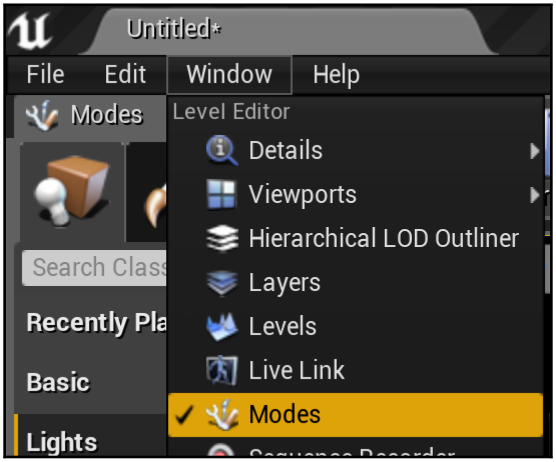
Just make sure you save the current scene first!

Adding light sources

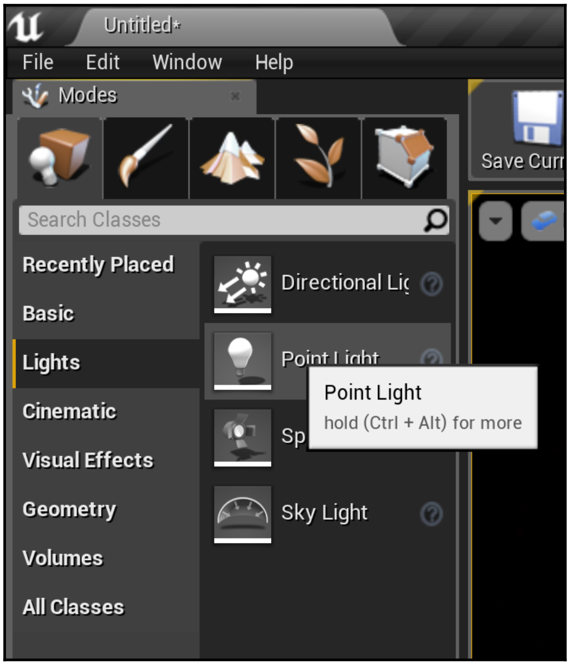
Note that your scene may appear completely (or mostly) black when you try to run it. This is because you haven't put a light source in it yet!

In the previous scene, the **P_Fire** particle emitter acts as a light source, but it only emits a small amount of light. To make sure that everything appears well lit in your scene, you should add a light source, as follows:

1. Go to **Window** and then click on **Modes** to ensure that the light sources panel is shown:



2. From the **Modes** panel, drag one of the **Lights** objects into the scene:



3. Select the **Lightbulb** and box icon (it looks like a mushroom, but it isn't).
4. Click on **Lights** in the left-hand panel.
5. Select the type of light you want and just pull it into your scene.

If you don't have a light source, your scene will appear completely black when you try to run it (or if there are just no objects in the scene).

Collision volumes

You might have noticed that, so far, the camera just passes through at least some of the scene geometry, even in play mode. That's not good. Let's make it so that the player can't just walk through the rocks in our scene.

There are a few different types of collision volumes. Generally, perfect mesh-mesh collisions are way too expensive to do at runtime. Instead, we use an approximation (a bounding volume) to guess the collision volume.

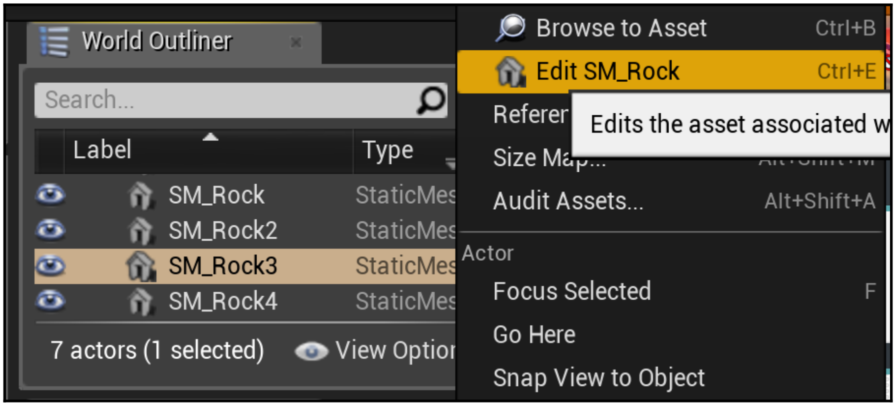
A mesh is the actual geometry of an object.

Adding collision volumes

The first thing we have to do is associate a collision volume with each of the rocks in the scene.

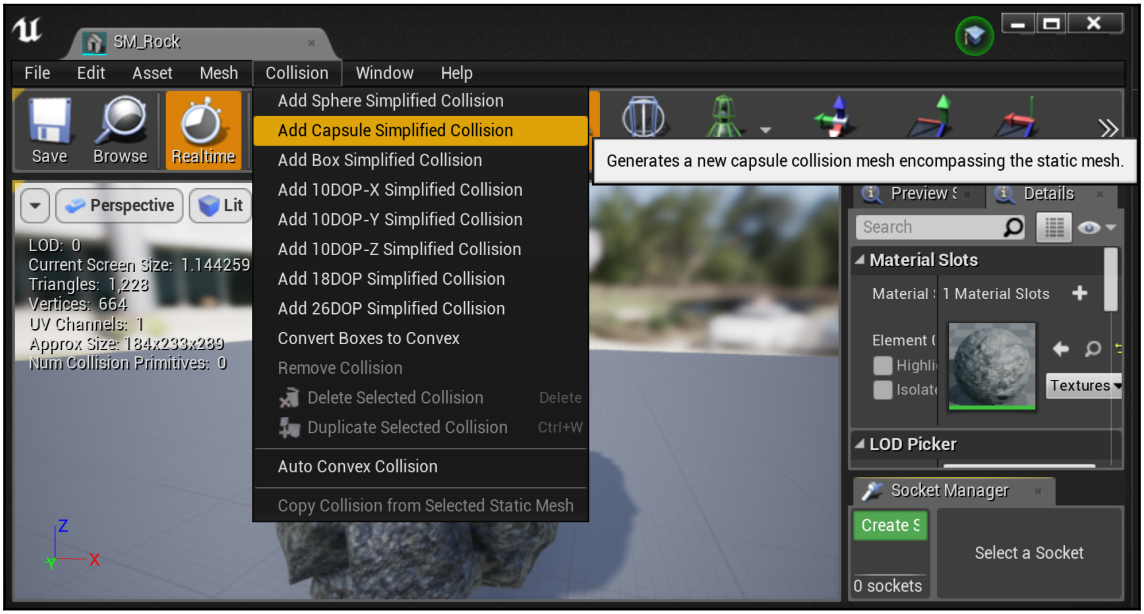
We can do this from the UE4 editor as follows:

1. Click on an object in the scene for which you want to add a collision volume.
2. Right-click on this object in the **World Outliner** tab (the default appears on the right-hand side of the screen) and select edit, as shown in the following screenshot:

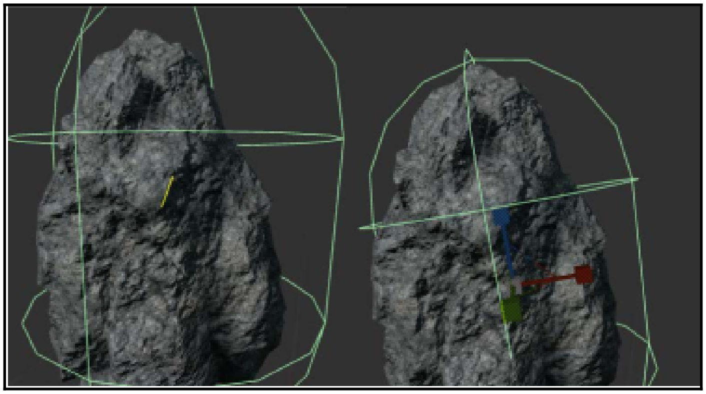


You will find yourself in the mesh editor.

3. Go to the **Collision** menu and then click on **Add Capsule Simplified Collision**:



4. The collision volume, when added successfully, will appear as a bunch of lines surrounding the object, as shown in the following screenshot:



The default collision capsule (left) and manually resized versions (right)

5. You can resize (R), rotate (E), move (W), and change the collision volume as you wish, the same way you would manipulate an object in the UE4 editor.
6. When you're done adding collision meshes, save and go back to the main editor window and click on **Play**; you will notice that you can no longer pass through your collidable objects.

Adding the player to the scene

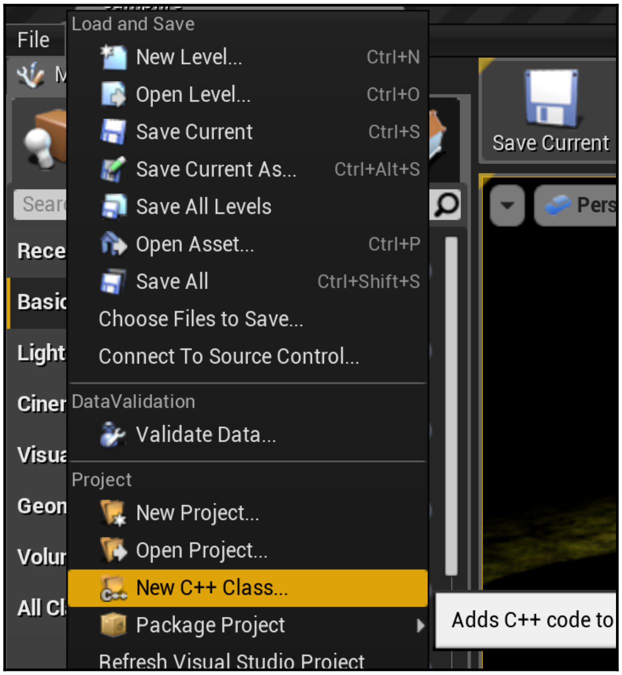
Now that we have a scene up and running, we need to add an actor to the scene. Let's first add an avatar for the player, complete with a collision volume. To do this, we'll have to inherit from a class from the UE4 `GameFramework` such as `Actor` or `Character`.

In order to create an onscreen representation of the player, we'll need to derive from the `ACharacter` class in Unreal.

Inheriting from UE4 GameFramework classes

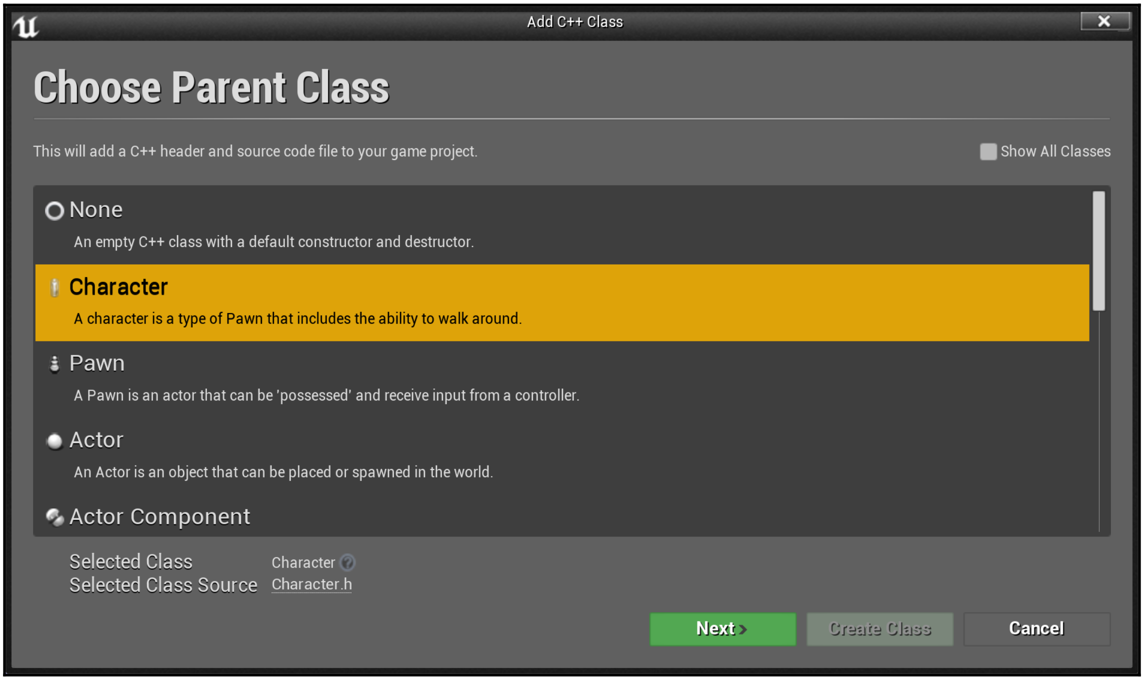
UE4 makes it easy to inherit from the base framework classes. All you have to do is perform the following steps:

1. Open your project in the UE4 editor.
2. Go to **File** and then select **New C++ Class...**:

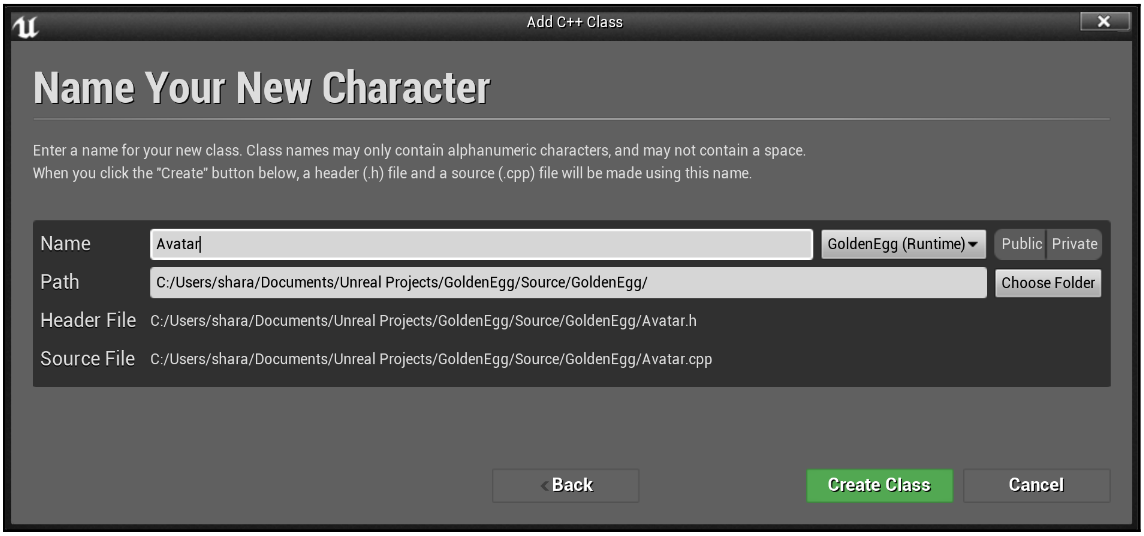


Navigating to File | New C++ Class... will allow you to derive from any of the UE4 GameFramework classes

3. Choose the base class you want to derive from. You have **Character**, **Pawn**, **Actor**, and so on, but for now, we will derive from **Character**:



4. Select the UE4 class you want to derive from.
5. Click on **Next** to get this dialog box, where you name the class. I named my player class `Avatar`:



6. Click on **Create Class** to create the class in code, as shown in the preceding screenshot.

Let UE4 refresh your Visual Studio or Xcode project if it asks you. Open the new `Avatar.h` file from the **Solution Explorer**.

The code that UE4 generates will look a little weird. Remember the macros that I suggested you avoid in `Chapter 5`, *Functions and Macros*? The UE4 code uses macros extensively. These macros are used to copy and paste boilerplate starter code that lets your code integrate with the UE4 editor.

The contents of the `Avatar.h` file are shown in the following code:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "Avatar.generated.h"

UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
    GENERATED_BODY()

public:
    // Sets default values for this character's properties
    AAvatar();
```

```
protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;

};
```

Let's talk about macros for a moment.

The `UCLASS()` macro basically makes your C++ code class available in the UE4 editor. The `GENERATED_BODY()` macro copies and pastes code that UE4 needs to make your class function properly as a UE4 class.

TIP

For `UCLASS()` and `GENERATED_BODY()`, you don't truly need to understand how UE4 works its magic. You just need to make sure that they are present at the right spot (where they were when you generated the class).

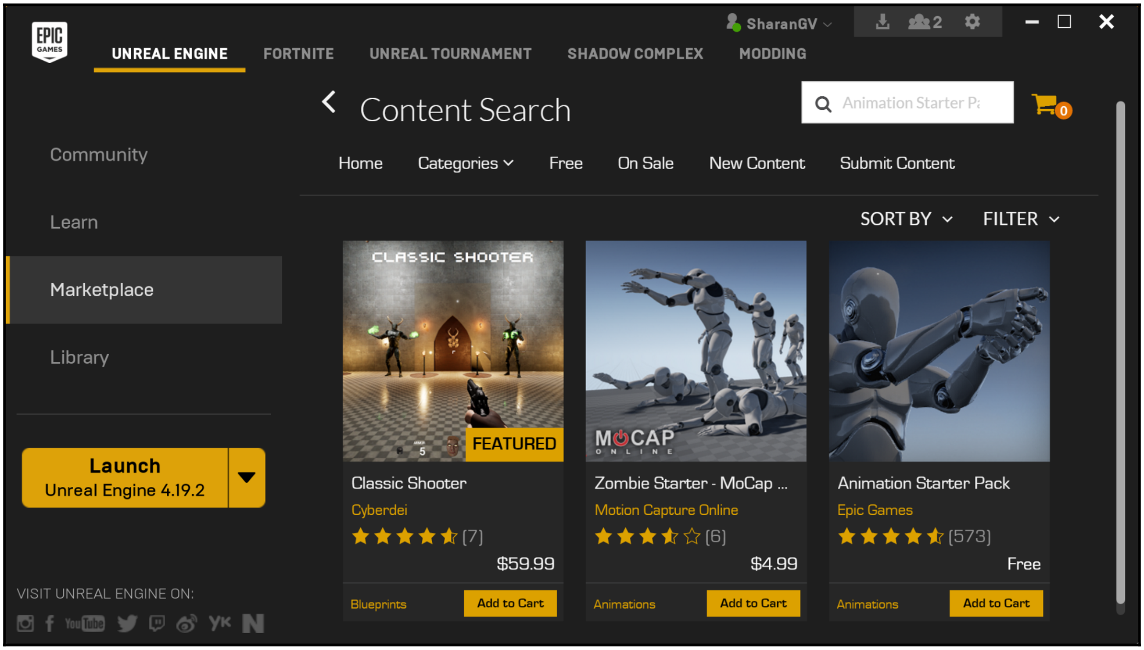
Associating a model with the Avatar class

Now, we need to associate a model with our character object. To do this, we need a model to play with. Fortunately, there is a whole pack of sample models available from the UE4 marketplace for free.

Downloading free models

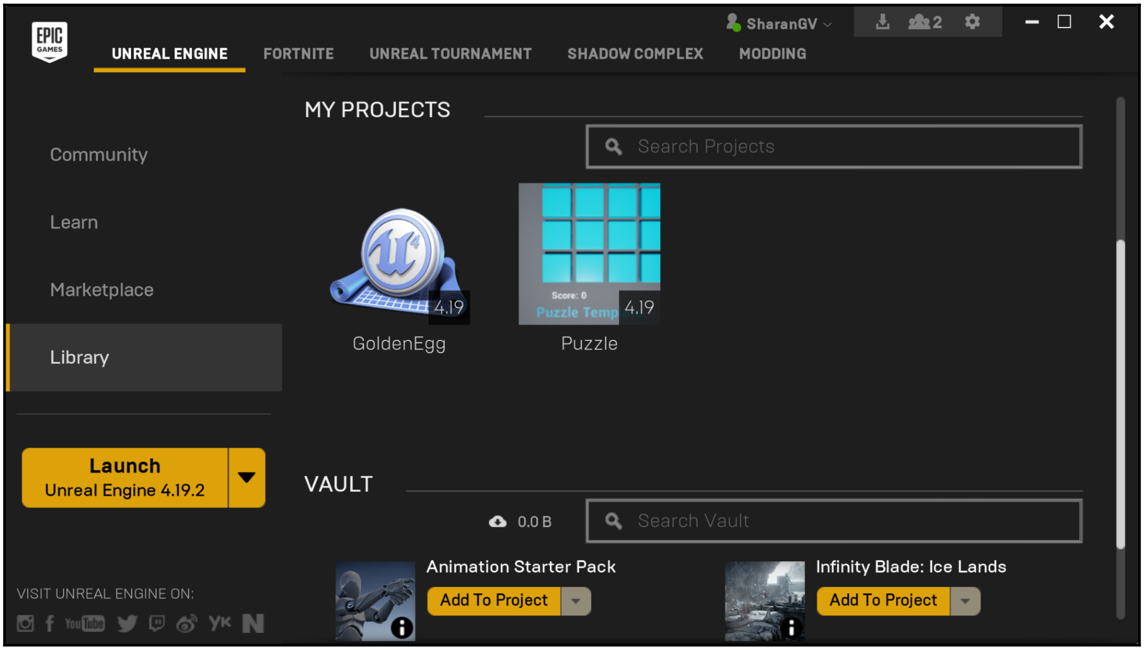
To create the player object, perform the following steps:

1. Download the **Animation Starter Pack** file (which is free) from the **Marketplace** tab. The easiest way to find it is to search for it:

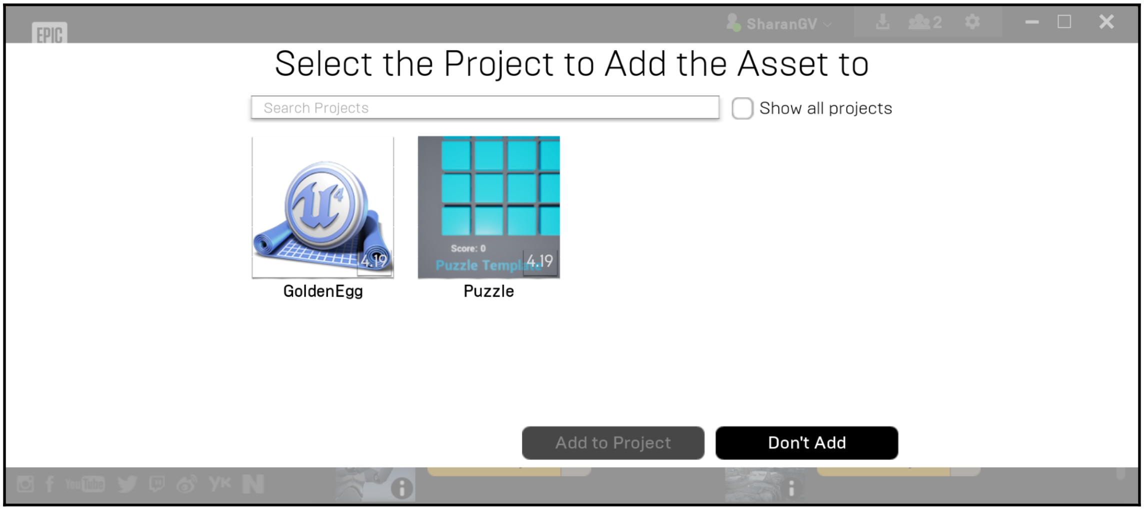


2. From the Unreal **Launcher**, click on **Marketplace** and search for **Animation Starter Pack**, which is free at the time of writing this book.

3. Once you've downloaded the **Animation Starter Pack** file, you will be able to add it to any of the projects you've previously created, as shown in the following screenshot:



4. When you click on **Add to project** under **Animation Starter Pack**, you will get this popup, asking which project to add the pack to:



5. Simply select your project and the new artwork will be available in your **Content Browser**.

Loading the mesh

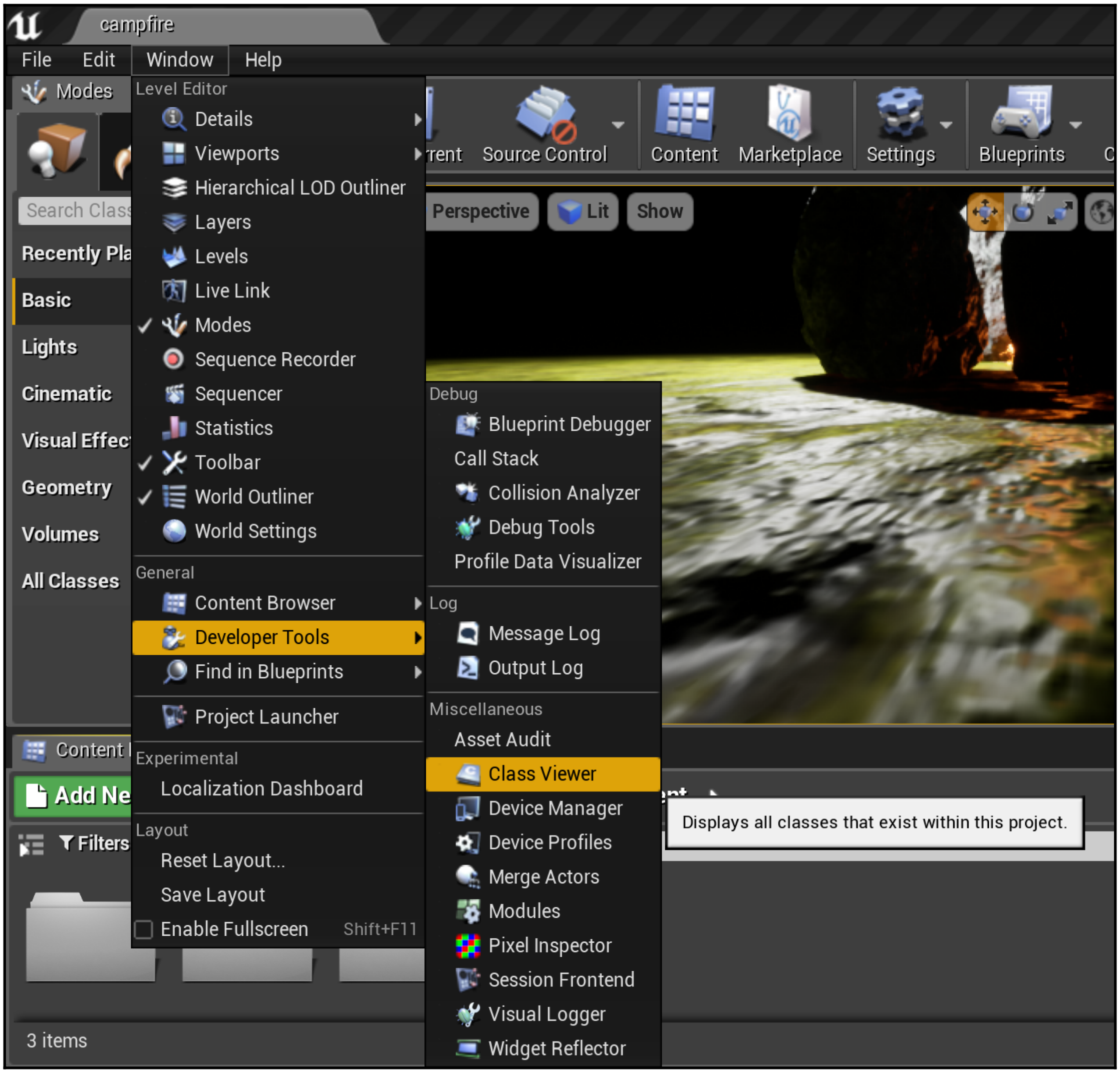
In general, it is considered a bad practice to hardcode your assets (or objects used in-game) into the game. Hardcoding means that you write C++ code that specifies the asset to load. However, hardcoding means the loaded asset is part of the final executable, which will mean that changing the asset that is loaded wouldn't be modifiable at runtime. This is a bad practice. It is much better to be able to change the asset loaded during runtime.

For this reason, we're going to use the UE4 blueprints feature to set up the model mesh and collision capsule of our `Avatar` class.

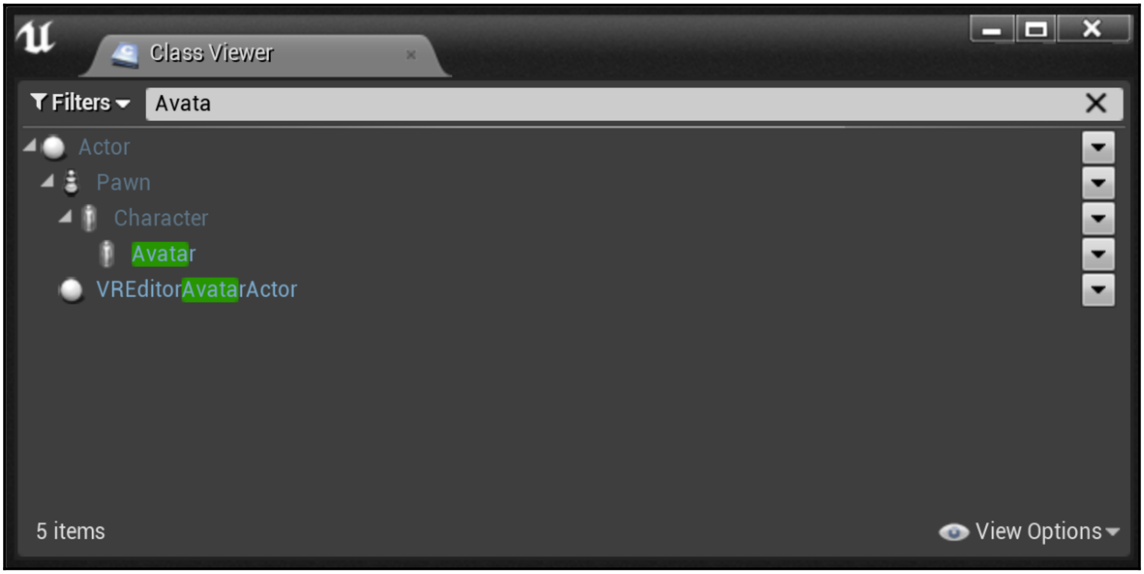
Creating a blueprint from our C++ class

Let's go ahead and create a blueprint—it's really easy:

1. Open the **Class Viewer** tab by navigating to **Window** | **Developer Tools** and then clicking on **Class Viewer**, as shown here:



2. In the **Class Viewer** dialog, start typing in the name of your C++ class. If you have properly created and exported the class from your C++ code, it will appear, as shown in the following screenshot:

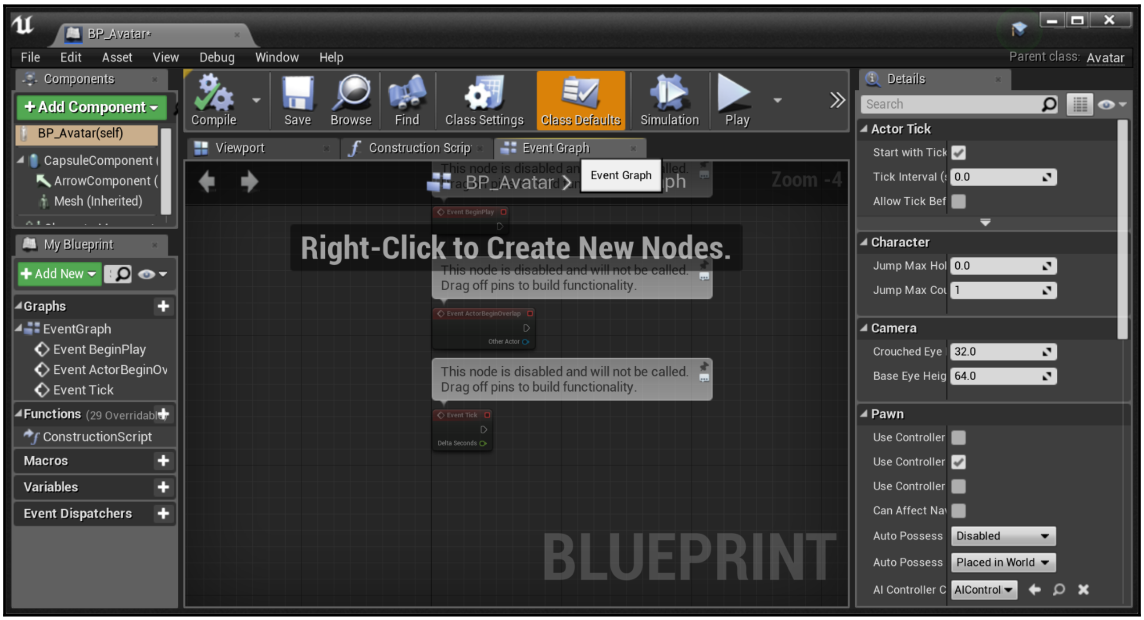


If your `Avatar` class does not show up, close the editor and compile/run the C++ project in Visual Studio or Xcode again.

3. Right-click on the class that you want to create a blueprint of (in my case, it's my **Avatar** class) and choose **Create Blueprint Class...**.
4. Name your blueprint something unique. I called my blueprint **BP_Avatar**. BP_ identifies it as a blueprint and makes it easier to search for later.

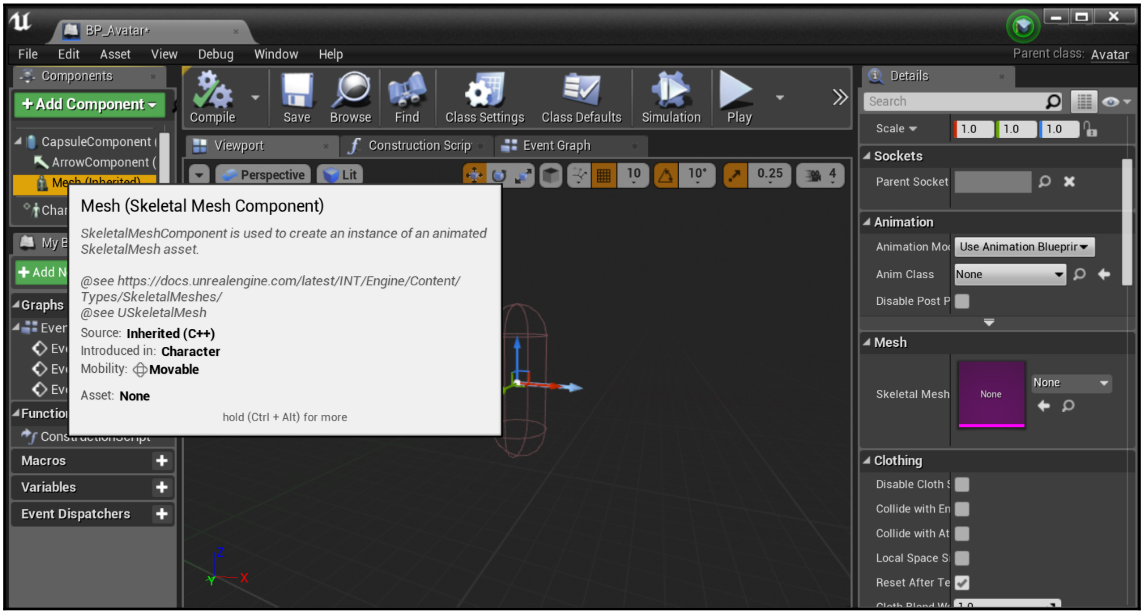
5. The new blueprint should open automatically for editing. If it doesn't, open it by double-clicking on **BP_Avatar** (it will appear in the **Class Viewer** tab after you add it, just under **Avatar**), as shown in the following screenshot:

6. You will be presented with the blueprints window for your new **BP_Avatar** object, as shown here (make sure to select the **Event Graph** tab):

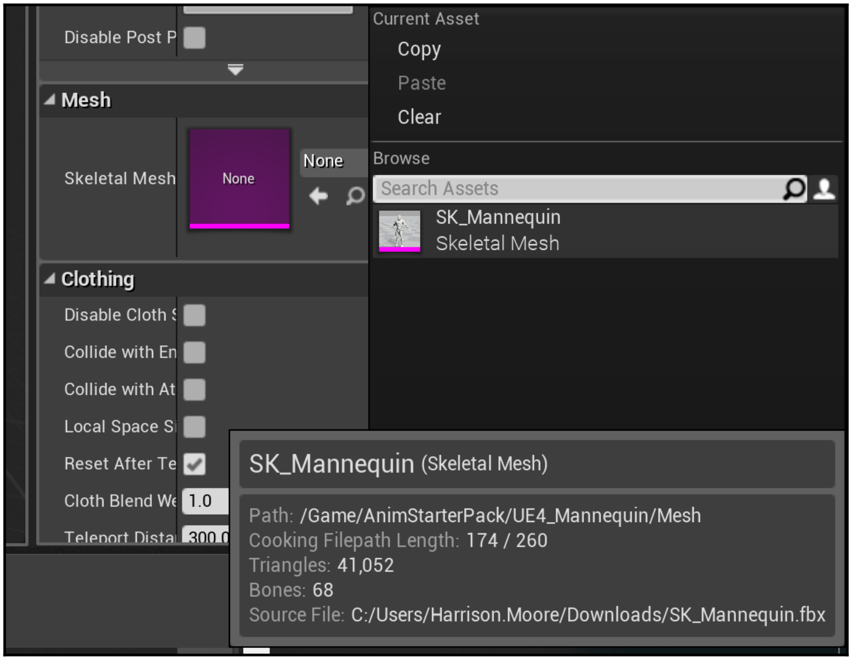


From this window, you can attach a model to the `Avatar` class visually. Again, this is the recommended pattern since artists will typically be the ones setting up their assets for game designers to play with.

7. Your blueprint will have already inherited a default skeletal mesh. To see the options for it, click on **Mesh (Inherited)** under **CapsuleComponent** on the left:

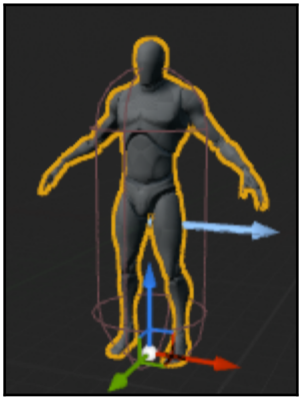


8. Click on the dropdown and select **SK_Mannequin** for your mesh:

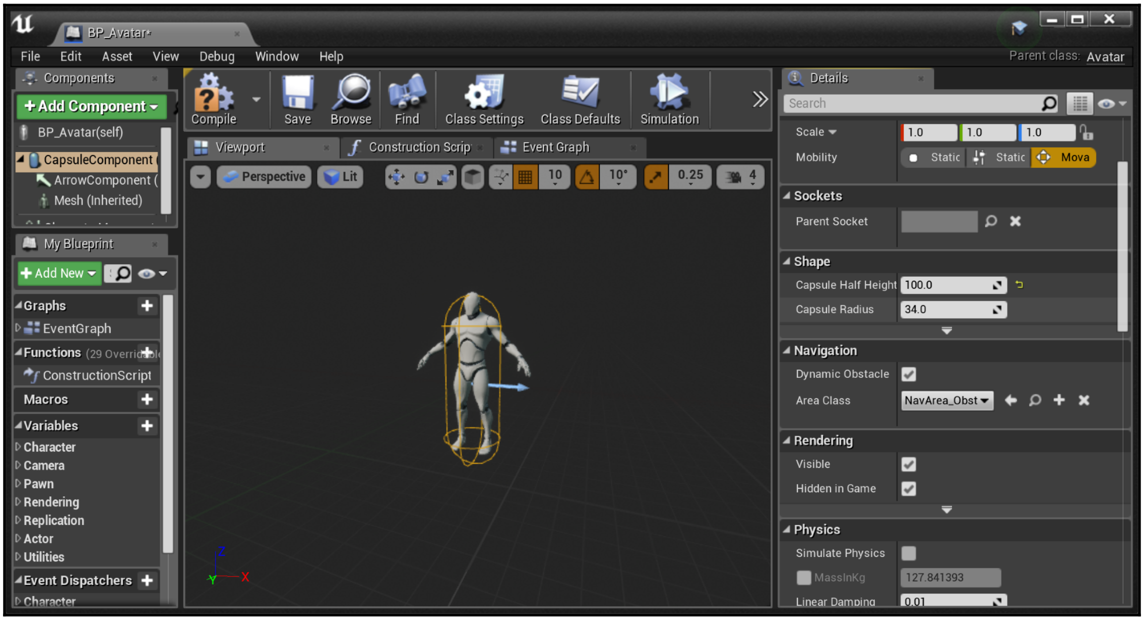


9. If **SK_Mannequin** doesn't appear in the dropdown, make sure that you download and add the **Animation Starter Pack** to your project.
10. What about the collision volume? You already have one called **CapsuleComponent**. If your capsule doesn't encapsulate your model, adjust the model so that it fits.

If your model ended up like mine, the capsule is off the mark! We need to adjust it.

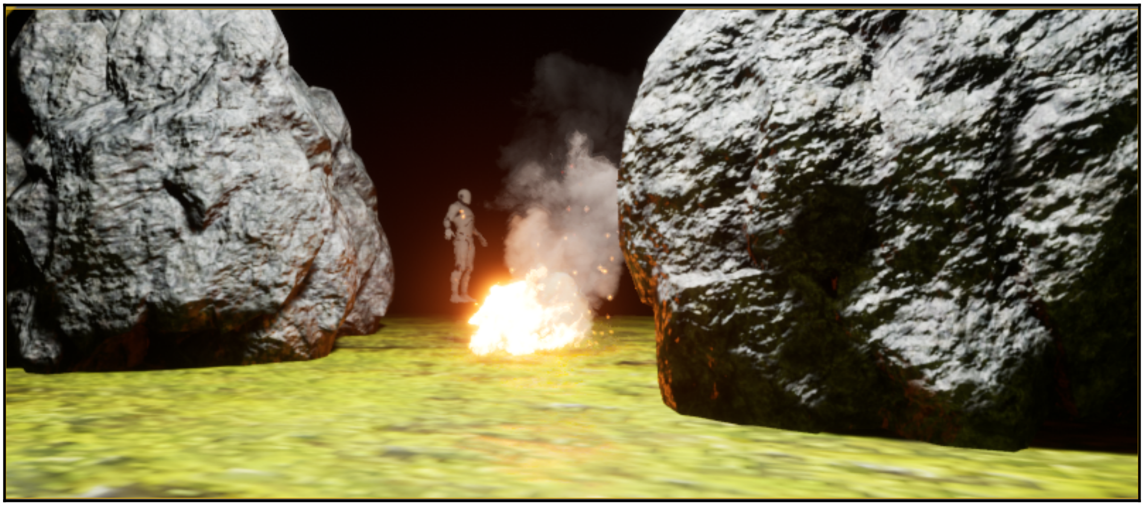


11. Click on the Avatar model and then click and hold the blue arrow pointing up, as shown in the preceding screenshot. Move him down until he fits inside the capsule. If the capsule isn't big enough, you can adjust its size in the **Details** tab under **Capsule Half-Height** and **Capsule Radius**:



You can stretch your capsule by adjusting the Capsule Half-Height property

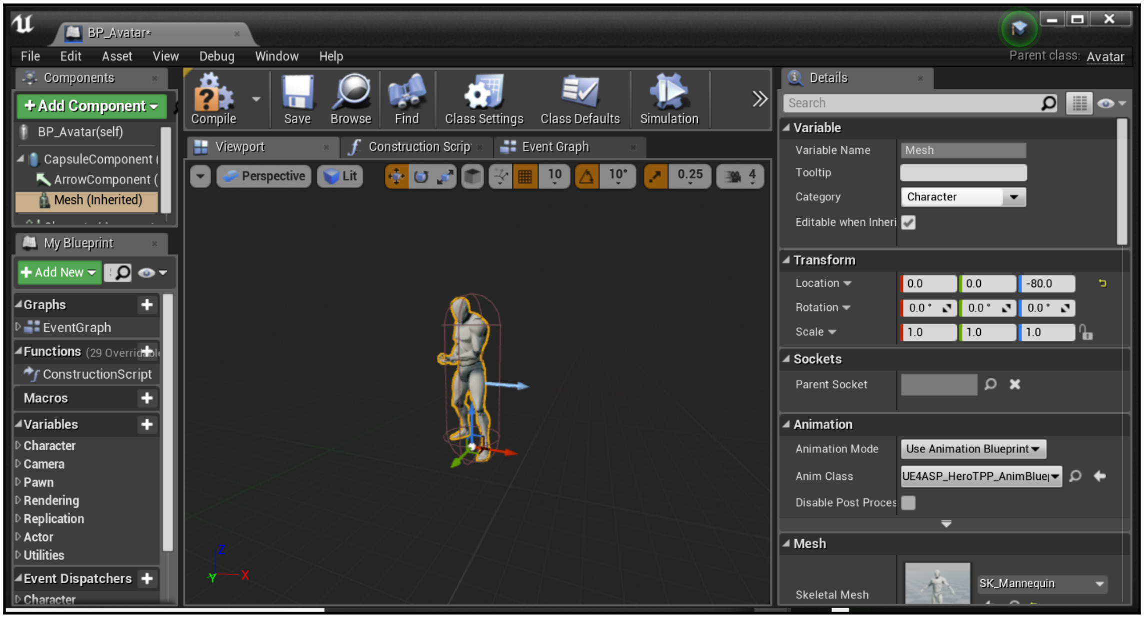
12. Let's add this avatar to the game world. Click and drag your **BP_Avatar** model from the **Class Viewer** tab to your scene in the UE4 editor:



Our Avatar class added to the scene

The pose of **Avatar** is the default pose. You want him animated, you say! Well, that's easy, just perform the following steps:

1. Click on your **Mesh** in the Blueprint editor and you will see **Animation** under **Details** on the right. Note: if you closed the blueprint for any reason and reopen it you won't see the full blueprint. If that happens, click the link to open the full blueprint editor.
2. You can now use a blueprint for the animation. This way, an artist can properly set the animation based on what the character is doing. If you select **UE4ASP_HeroTPP_AnimBlueprint** from the `AnimClass` drop-down menu, the animation will be adjusted by the blueprint (which would have been done by an artist) as the character moves:



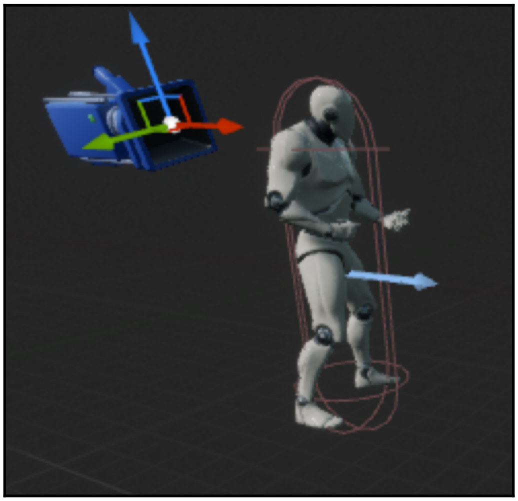
If you save and compile the blueprint and hit play in the main game window, you will see the idle animation.

> **TIP**
> We can't cover everything here. Animation blueprints are covered in `Chapter 11`, *Monsters*. If you're really interested in animation, it wouldn't be a bad idea to sit through a couple of Gnomon Workshop tutorials on IK, animation, and rigging, which can be found at `gnomonworkshop.com/tutorials`.

One more thing: let's make the camera for the Avatar appear behind it. This will give you a third-person point of view, which will allow you to see the whole character, shown in the following screenshot, with the corresponding steps:

1. In the **BP_Avatar** blueprint editor, select **BP_Avatar (Self)** and click on **Add Component**.
2. Scroll down to choose to add a **Camera**.

A camera will appear in the viewport. You can click on the camera and move it around. Position the camera so that it is somewhere behind the player. Make sure that the blue arrow on the player is facing the same direction as the camera. If it isn't, rotate the Avatar model mesh so that it faces the same direction as its blue-colored arrow:



The blue-colored arrow on your model mesh indicates the forward direction for the model mesh. Make sure that the camera's opening faces the same direction as the character's forward vector.

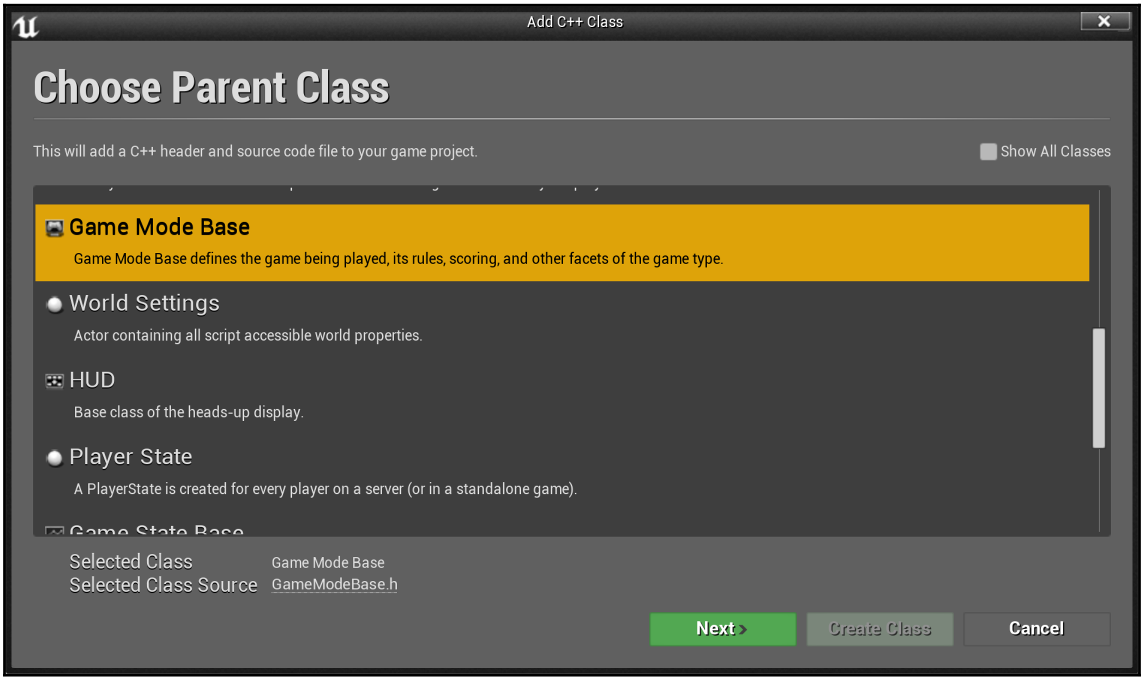
Writing C++ code that controls the game's character

When you launch your UE4 game, you might notice that the camera hasn't changed. What we will do now is make the starting character an instance of our `Avatar` class and control our character using the keyboard.

Making the player an instance of the Avatar class

Let's take a look at how we go about this. In the Unreal Editor, perform the following steps:

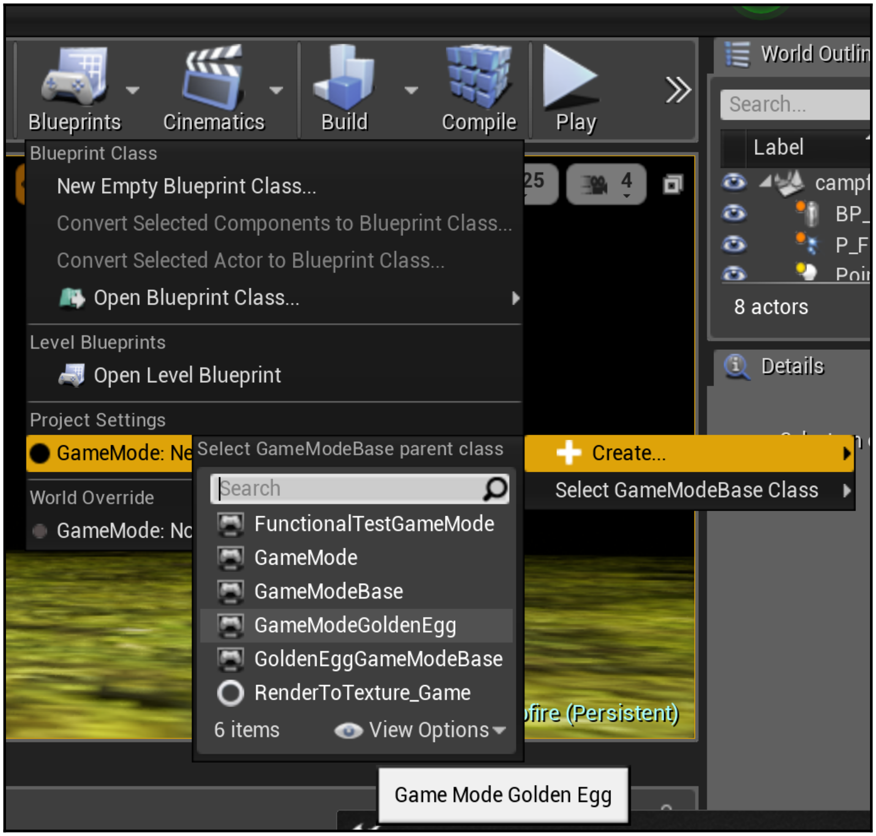
1. Create a subclass of **Game Mode** by navigating to **File** | **New C++ Class...** and selecting **Game Mode Base**. I named mine `GameModeGoldenEgg`:



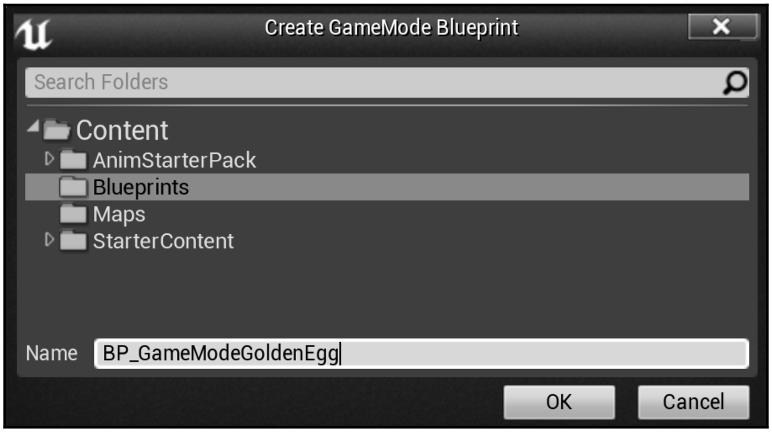
The UE4 **GameMode** contains the rule of the game and describes how the game is played to the engine. We will work more with our `GameMode` class later. For now, we need to subclass it.

It should automatically compile your C++ code after you create the class, so you can create a `GameModeGoldenEgg` blueprint.

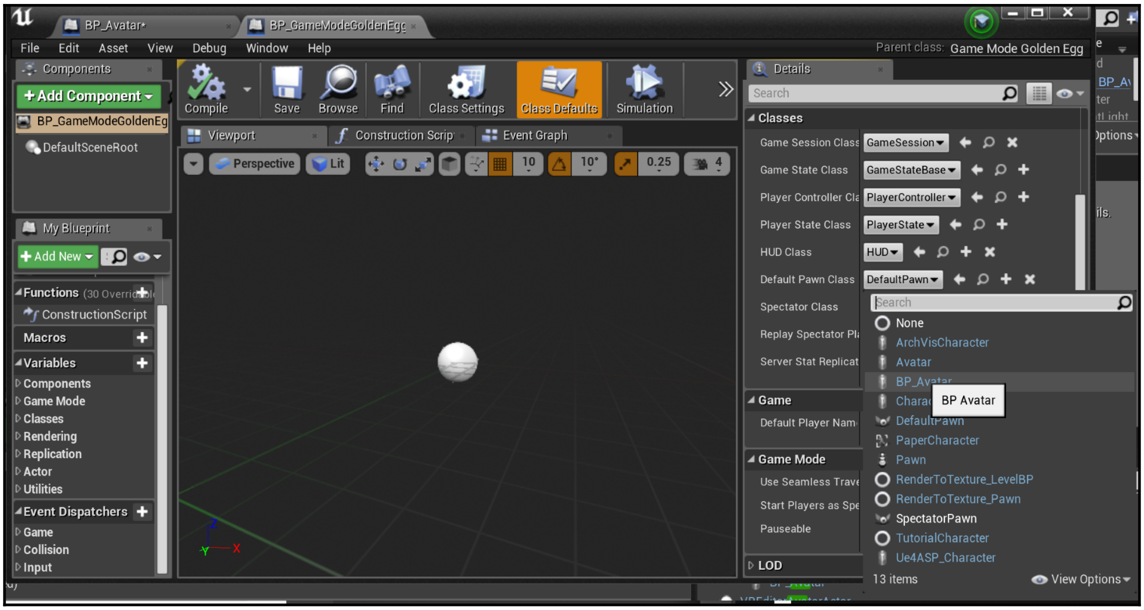
2. Create the **GameMode** blueprint by going to the **Blueprints** icon in the menu bar at the top, clicking on **GameMode New**, and then selecting **+ Create** | **GameModeGoldenEgg** (or whatever you named your **GameMode** subclass in step 1):



3. Name your blueprint; I called mine `BP_GameModeGoldenEgg`:



4. Your newly created blueprint will open in the blueprint editor. If it doesn't, you can open the **BP_GameModeGoldenEgg** class from the **Class Viewer** tab.
5. Select your **BP_Avatar** class from the **Default Pawn Class** panel, as shown in the following screenshot. The **Default Pawn Class** panel is the type of object that will be used for the player:



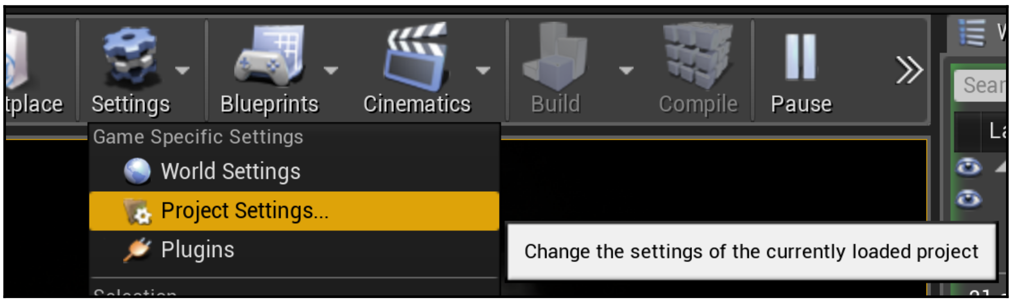
6. Launch your game. You can see a back view as the camera is placed behind the player:

You'll notice that you can't move. Why is that? The answer is because we haven't set up the controller input yet. The following section will teach you exactly how to go about doing it.

Setting up controller input

Here are the steps to set up input:

1. To set up controller input, go to **Settings** | **Project Settings...**:



2. In the left-hand panel, scroll down until you see **Input** under **Engine**:

3. On the right-hand side, you can set up some **Bindings**. Click **+** to add a new binding and then click on the small arrow next to **Axis Mappings** in order to expand it. Add just two axis mappings to start, one called **Forward** (connected to the keyboard letter *W*) and one called **Strafe** (connected to the keyboard letter *D*). Remember the names that you set; we will look them up in C++ code in just a moment.
4. Close the **Project Settings** dialog. Open your C++ code. In the `Avatar.h` constructor, you need to add two member function declarations, as shown here:

```
UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
    GENERATED_BODY()

public:
    // Sets default values for this character's properties
    AAvatar();

protected:
    // Called when the game starts or when spawned
```

```
        virtual void BeginPlay() override;

    public:
        // Called every frame
        virtual void Tick(float DeltaTime) override;

        // Called to bind functionality to input
        virtual void SetupPlayerInputComponent(class
    UInputComponent* PlayerInputComponent) override;

        // New! These 2 new member function declarations
        // they will be used to move our player around!
        void MoveForward(float amount);
        void MoveRight(float amount);
    };
```

Notice how the existing functions, `SetupPlayerInputComponent` and `Tick`, are overrides of virtual functions. `SetupPlayerInputComponent` is a virtual function in the `APawn` base class. We will also be adding code to this function.

5. In the `Avatar.cpp` file, you need to add the function bodies. Inside `SetupPlayerInputComponent` under `Super::SetupPlayerInputComponent(PlayerInputComponent);`, add the following lines:

```
  check(PlayerInputComponent);
    PlayerInputComponent->BindAxis("Forward", this,
        &AAvatar::MoveForward);
    PlayerInputComponent->BindAxis("Strafe", this,
&AAvatar::MoveRight);
```

This member function looks up the **Forward** and **Strafe** axis bindings that we just created in Unreal Editor and connects them to the member functions inside the `this` class. Which member functions should we connect to? Why, we should connect to `AAvatar::MoveForward` and `AAvatar::MoveRight`. Here are the member function definitions for these two functions:

```
void AAvatar::MoveForward( float amount )
{
  // Don't enter the body of this function if Controller is
  // not set up yet, or if the amount to move is equal to 0
  if( Controller && amount )
  {
    FVector fwd = GetActorForwardVector();
    // we call AddMovementInput to actually move the
    // player by `amount` in the `fwd` direction
```

```
    AddMovementInput(fwd, amount);
  }
}

void AAvatar::MoveRight( float amount )
{
  if( Controller && amount )
  {
    FVector right = GetActorRightVector();
    AddMovementInput(right, amount);
  }
}
```

The `Controller` object and the `AddMovementInput` function are defined in the `APawn` base class. Since the `Avatar` class derives from `ACharacter`, which in turn derives from `APawn`, we get free use of all the member functions in the `APawn` base class. Now, do you see the beauty of inheritance and code reuse? If you test this out, make sure you click inside the game window, because otherwise the game won't receive keyboard events.

Exercise

Add axis bindings and C++ functions to move the player to the left and back.

Here's a hint: you only need to add axis bindings if you realize going backward is simply the negative of going forward.

Solution

Enter two extra axis bindings by navigating to **Settings** | **Project Settings...** | **Input**, as shown here:

Scale the **S** and **A** input by -1.0. This will invert the axis, so pressing the *S* key in the game will move the player forward. Try it!

Alternatively, you can define two completely separate member functions in your `AAvatar` class, as follows, and bind the *A* and *S* keys to `AAvatar::MoveLeft` and `AAvatar::MoveBack`, respectively (and make sure to add the bindings for these to `AAvatar::SetupPlayerInputComponent`):

```
void AAvatar::MoveLeft( float amount )
{
  if( Controller && amount )
  {
    FVector left = -GetActorRightVector();
    AddMovementInput(left, amount);
  }
}
void AAvatar::MoveBack( float amount )
{
```

```
    if( Controller && amount )
    {
      FVector back = -GetActorForwardVector();
      AddMovementInput(back, amount);
    }
  }
```

Yaw and pitch

We can change the direction in which the player looks by setting the yaw and pitch of the controller. Check out the following steps:

1. Add new axis bindings for the mouse, as shown in the following screenshot:

2. From C++, add two new member function declarations to `AAvatar.h`:

```
void Yaw( float amount );
void Pitch( float amount );
```

The bodies of these member functions will go in the `AAvatar.cpp` file:

```
void AAvatar::Yaw(float amount)
{
    AddControllerYawInput(200.f * amount *
GetWorld()->GetDeltaSeconds());
}
void AAvatar::Pitch(float amount)
{
    AddControllerPitchInput(200.f * amount *
GetWorld()->GetDeltaSeconds());
}
```

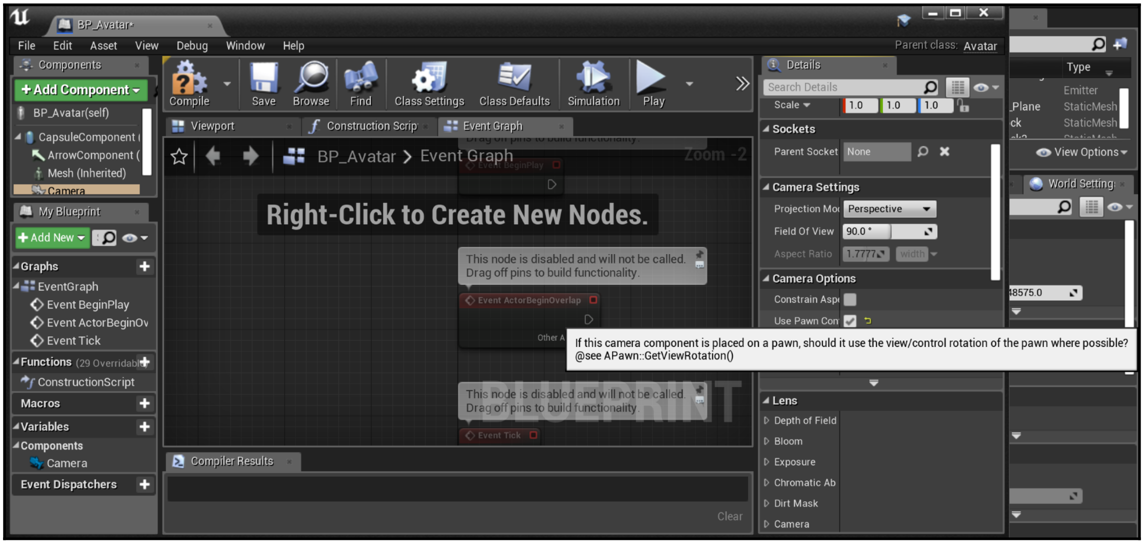
3. Add two lines to `SetupPlayerInputComponent`:

```
void AAvatar::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
{
  // .. as before, plus:
  PlayerInputComponent->BindAxis("Yaw", this, &AAvatar::Yaw);
  PlayerInputComponent->BindAxis("Pitch", this, &AAvatar::Pitch);
}
```

Here, notice how I've multiplied the `amount` values in the `Yaw` and `Pitch` functions by 200. This number represents the mouse's sensitivity. You can (should) add a `float` member to the `AAvatar` class in order to avoid hardcoding this sensitivity number.

`GetWorld()->GetDeltaSeconds()` gives you the amount of time that passed between the last frame and this frame. It isn't a lot; `GetDeltaSeconds()` should be around 16 milliseconds (0.016 s) most of the time (if your game is running at 60 fps).

Note: you may notice that right now Pitch doesn't actually work. This is because you're using a third-person camera. While it might not make sense for this camera, you can get it working by going into **BP_Avatar**, selecting the camera, and checking **Use Pawn Control Rotation** under **Camera Options**:



So, now we have player input and control. To add new functionality to your Avatar, this is all that you have to do:

1. Bind your key or mouse actions by going to **Settings** | **Project Settings** | **Input**.
2. Add a member function to run when that key is pressed.
3. Add a line to `SetupPlayerInputComponent`, connecting the name of the bound input to the member function we want to run when that key is pushed.

Creating non-player character entities

So, we need to create a few **NPC** (**non-playable characters**). NPCs are characters in the game that help the player. Some offer special items, some are shop vendors, and some have information to give to the player. In this game, they will react to the player as he gets near. Let's program in some of this behavior:

1. Create another subclass of **Character**. In the UE4 Editor, go to **File** | **New C++ Class...** and choose the **Character** class from which you can make a subclass. Name your subclass `NPC`.
2. Edit your code in Visual Studio. Each NPC will have a message to tell the player, so we add in a `UPROPERTY() FString` property to the `NPC` class.

> **TIP:** `FString` is UE4's version of C++'s `<string>` type. When programming in UE4, you should use `FString` objects over C++ STL's `string` objects. In general, you should use UE4's built-in types, as they guarantee cross-platform compatibility.

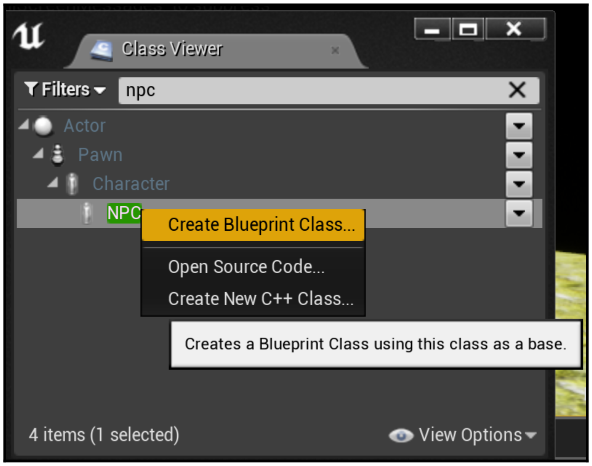
3. Here's how to add the `UPROPERTY() FString` property to the `NPC` class:

```
UCLASS()
class GOLDENEGG_API ANPC : public ACharacter
{
    GENERATED_BODY()
    // This is the NPC's message that he has to tell us.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        NPCMessage)
        FString NpcMessage;
    // When you create a blueprint from this class, you want to be
    // able to edit that message in blueprints,
    // that's why we have the EditAnywhere and BlueprintReadWrite
    // properties.
public:
    // Sets default values for this character's properties
    ANPC();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;
};
```

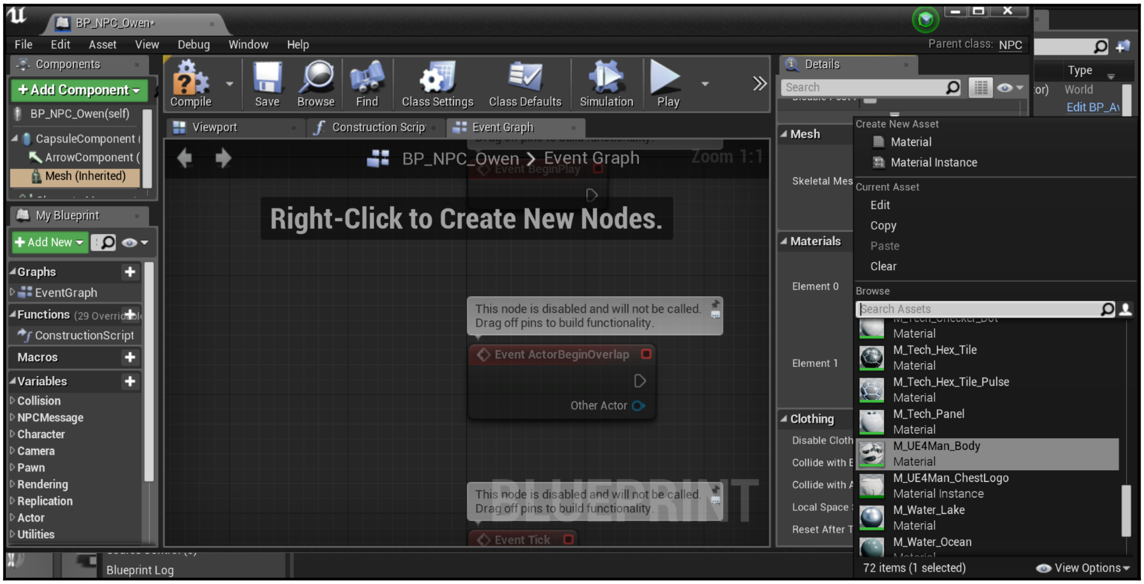
Notice that we put the `EditAnywhere` and `BlueprintReadWrite` properties into the `UPROPERTY` macro. This will make `NpcMessage` editable in blueprints.

> **TIP** Full descriptions of all the UE4 property specifiers are available at `https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Reference/Properties/index.html`.

4. Recompile your project (as we did for the `Avatar` class). Then, go to the **Class Viewer**, right-click on your `NPC` class, and create a blueprint class from it.
5. Each NPC character you want to create can be a blueprint based off of the `NPC` class. Name each blueprint something unique, as we'll be selecting a different model mesh and message for each NPC that appears, as shown in the following screenshot:

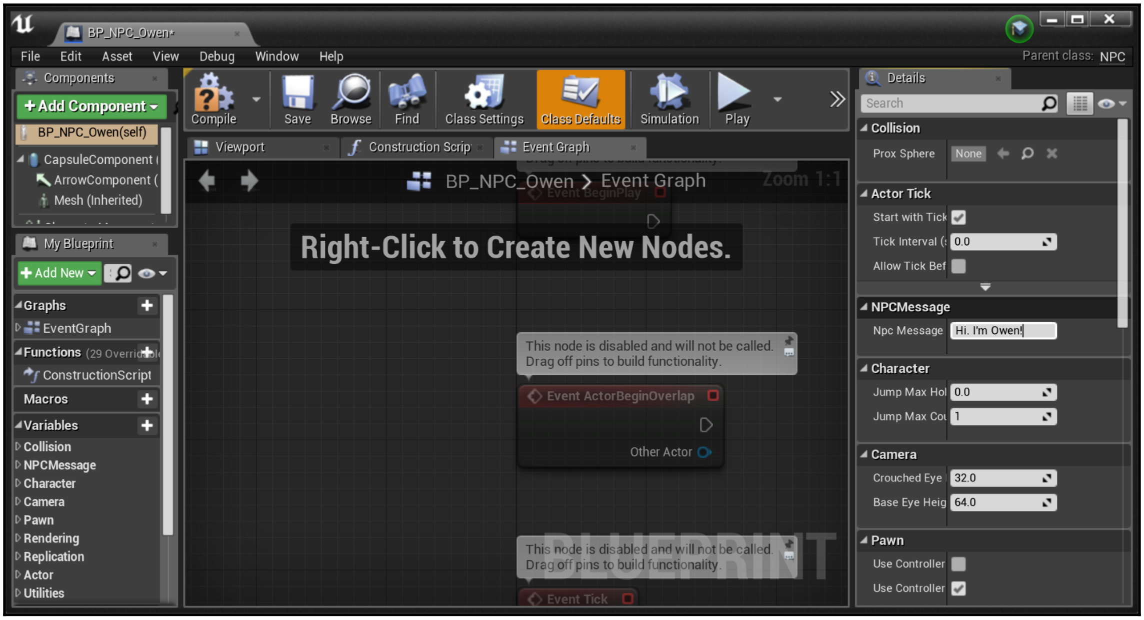


6. Open the blueprint and select **Mesh (Inherited)**. You can then change the material of your new character in the **Skeletal Mesh** dropdown so that it looks different from the player:



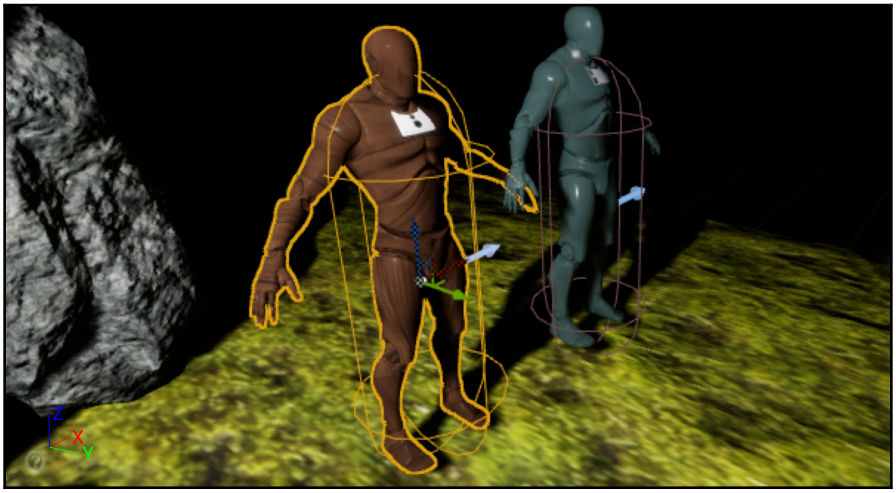
Change the material of your character in your mesh's properties by selecting from the dropdown for each element available

7. In the **Details** tab with the blueprint name (self) selected in the **Components** tab, look for the `NpcMessage` property. This is our connection between C++ code and blueprints; because we entered a `UPROPERTY()` function on the `FString NpcMessage` variable, that property appears editable in UE4, as shown in the following screenshot:



8. Drag **BP_NPC_Owen** into the scene. You can create a second or third character as well, and be sure to give them unique names, appearances, and messages:

I've created two blueprints for NPCs based on the NPC base classes: BP_NPC_Jonathan and BP_NPC_Owen. They have different appearances and different messages for the player:



Jonathan and Owen in the scene

Displaying a quote from each NPC dialog box

To display a dialog box, we need a custom **heads-up display** (**HUD**). In the UE4 editor, go to **File** | **New C++ Class...** and choose the `HUD` class from which the subclass is created (you'll need to scroll down to find it). Name your subclass as you wish; I've named mine `MyHUD`.

After you have created the `MyHUD` class, let Visual Studio reload. We will make some code edits.

Displaying messages on the HUD

Inside the `AMyHUD` class, we need to implement the `DrawHUD()` function in order to draw our messages to the HUD and to initialize a font draw to the HUD with, as shown in the following code in `MyHUD.h`:

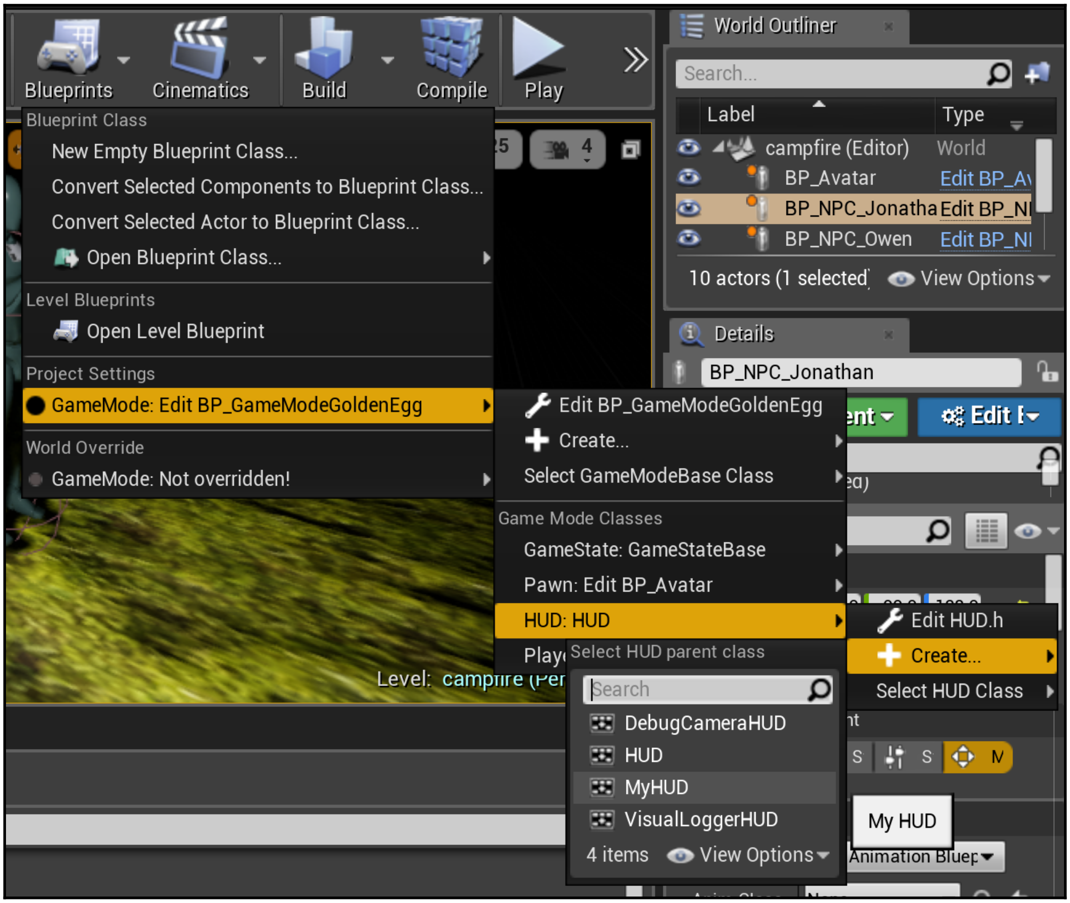
```
UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
    GENERATED_BODY()
public:
    // The font used to render the text in the HUD.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
    UFont* hudFont;
    // Add this function to be able to draw to the HUD!
    virtual void DrawHUD() override;
};
```

The HUD font will be set in a blueprinted version of the `AMyHUD` class. The `DrawHUD()` function runs once per frame. In order to draw within the frame, add a function to the `AMyHUD.cpp` file:

```
void AMyHUD::DrawHUD()
{
    // call superclass DrawHUD() function first
    Super::DrawHUD();
    // then proceed to draw your stuff.
    // we can draw lines..
    DrawLine(200, 300, 400, 500, FLinearColor::Blue);
    // and we can draw text!
    const FVector2D ViewportSize =
FVector2D(GEngine->GameViewport->Viewport->GetSizeXY());
    DrawText("Greetings from Unreal!", FLinearColor::White,
ViewportSize.X/2, ViewportSize.Y/2, hudFont);
}
```

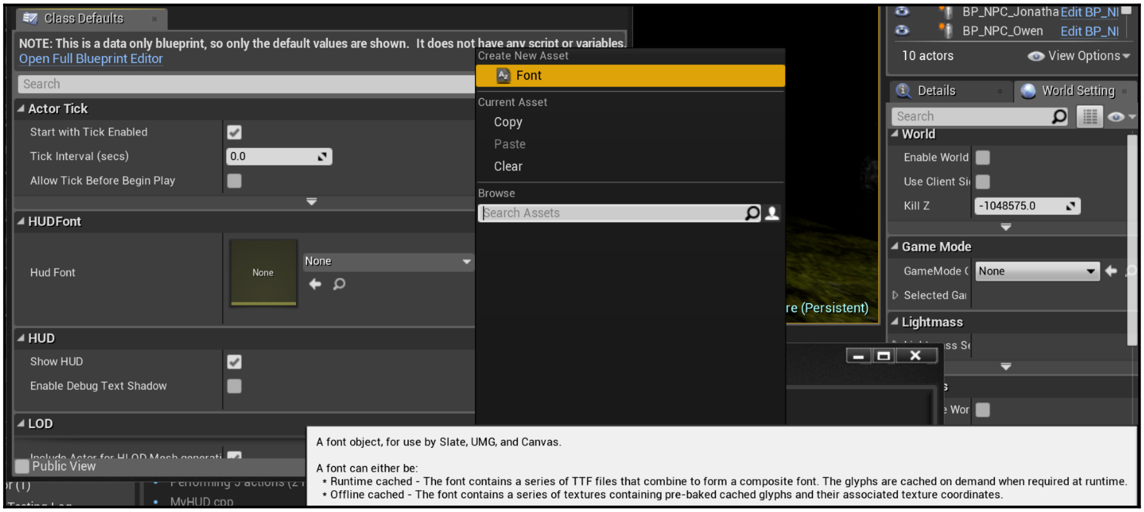
Wait! We haven't initialized our font yet. Let's do that now:

1. Set it up in blueprints. Compile your Visual Studio project in the editor, then go to the **Blueprints** menu at the top and navigate to **GameMode** | **HUD** | **+ Create** | **MyHUD**:

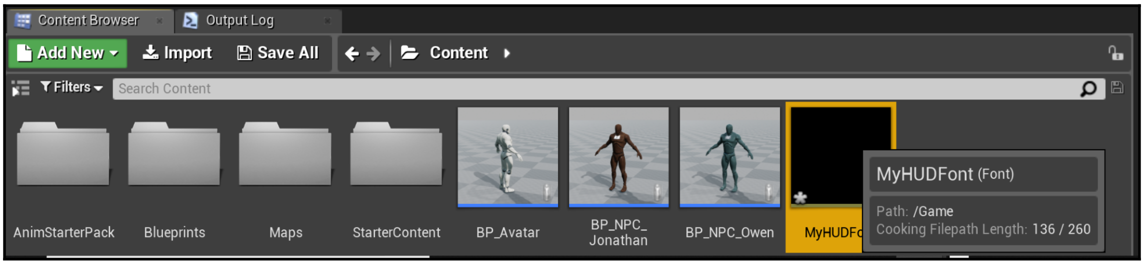


Creating a blueprint of the MyHUD class

2. I called mine `BP_MyHUD`. Find `Hud Font`, select the dropdown, and create a new Font asset. I named mine `MyHUDFont`:

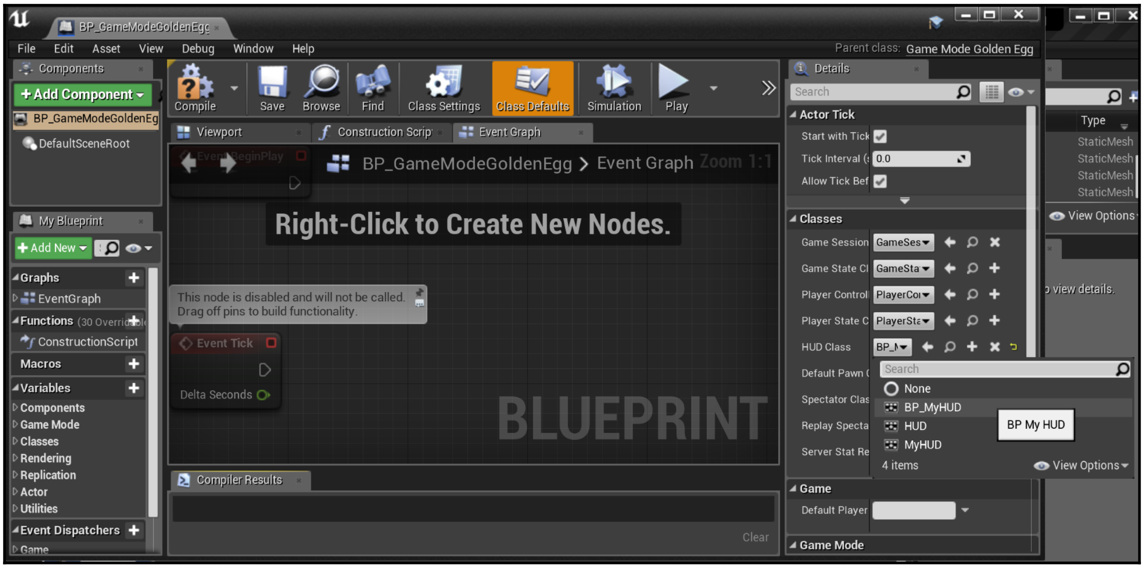


3. Locate MyHUDFont in the content browser and double-click on it to edit it:



In the window that follows, you can click on where it says `+ Add Font` to create a new Default Font Family. You can name it what you like and click the folder icon to choose a font from your hard drive (you can find .TTF or TrueType Fonts online for free at many sites – I used a Blazed font I found); When you import a font, it will ask you to save the Font Face. You will also want to change the Legacy Font Size in MyHUDFont to a much bigger size (I used 36).

4. Edit your **Game Mode** blueprint (**BP_GameModeGoldenEgg**) and select your new `BP_MyHUD` (not `MyHUD`) class for the **HUD Class** panel:



Compile and test your program by running it! You should see text printed on the screen:

Exercise

You can see that the text isn't fully centered. That's because the position is based on the top-left corner of the text, not the middle.

See whether you can fix that. Here's a hint: get the width and height of the text and subtract half of that from the viewport width and height/2 you're already using. You'll want to use something similar to the following:

```
    const FVector2D ViewportSize =
FVector2D(GEngine->GameViewport->Viewport->GetSizeXY());
    const FString message("Greetings from Unreal!");
    float messageWidth = 0;
    float messageHeight = 0;
    GetTextSize(message, messageWidth, messageHeight, hudFont);
    DrawText(message, FLinearColor::White, (ViewportSize.X - messageWidth)
/ 2, (ViewportSize.Y - messageHeight) / 2, hudFont);
```

Using TArray<Message>

Each message we want to display for the player will have a few properties:

- An `FString` variable for the message
- A `float` variable for the time to display it
- An `FColor` variable for the color of the message

So, it makes sense for us to write a little `struct` function to contain all this information.

At the top of `MyHUD.h`, insert the following `struct` declaration:

```
struct Message
{
  FString message;
  float time;
  FColor color;
  Message()
  {
    // Set the default time.
    time = 5.f;
    color = FColor::White;
  }
  Message( FString iMessage, float iTime, FColor iColor )
  {
    message = iMessage;
    time = iTime;
    color = iColor;
  }
};
```

Now, inside the `AMyHUD` class, we want to add a `TArray` of these messages. `TArray` is a UE4-defined special type of dynamically-growable C++ array. We will cover the detailed use of `TArray` in `Chapter 9`, *Templates and Commonly-Used Containers*, but this simple use of `TArray` should be a nice introduction to garner your interest in the usefulness of arrays in games. This will be declared as `TArray<Message>`:

```
UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
    GENERATED_BODY()
public:
    // The font used to render the text in the HUD.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
        UFont* hudFont;
    // New! An array of messages for display
    TArray<Message> messages;
    virtual void DrawHUD() override;
    // New! A function to be able to add a message to display
    void addMessage(Message msg);
};
```

Also add `#include "CoreMinimal.h"` *to the top of the file.*

Now, whenever the NPC has a message to display, we just need to call `AMyHud::addMessage()` with our message. The message will be added to the `TArray` of the messages to be displayed. When a message expires (after a certain amount of time), it will be removed from the HUD.

Inside the `AMyHUD.cpp` file, add the following code:

```
void AMyHUD::DrawHUD()
{
    Super::DrawHUD();
    // iterate from back to front thru the list, so if we remove
    // an item while iterating, there won't be any problems
    for (int c = messages.Num() - 1; c >= 0; c--)
    {
        // draw the background box the right size
        // for the message
        float outputWidth, outputHeight, pad = 10.f;
        GetTextSize(messages[c].message, outputWidth, outputHeight,
            hudFont, 1.f);

        float messageH = outputHeight + 2.f*pad;
        float x = 0.f, y = c * messageH;

        // black backing
```

```
            DrawRect(FLinearColor::Black, x, y, Canvas->SizeX, messageH
            );
            // draw our message using the hudFont
            DrawText(messages[c].message, messages[c].color, x + pad, y +
                pad, hudFont);

            // reduce lifetime by the time that passed since last
            // frame.
            messages[c].time -= GetWorld()->GetDeltaSeconds();

            // if the message's time is up, remove it
            if (messages[c].time < 0)
            {
                messages.RemoveAt(c);
            }
        }
    }

    void AMyHUD::addMessage(Message msg)
    {
        messages.Add(msg);
    }
```

The `AMyHUD::DrawHUD()` function now draws all the messages in the `messages` array, and arranges each message in the `messages` array by the amount of time that passed since the last frame. Expired messages are removed from the `messages` collection once their `time` value drops below 0.

Exercise

Refactor the `DrawHUD()` function so that the code that draws the messages to the screen is in a separate function, called `DrawMessages()`. You will probably want to create at least one sample message object and call `addMessage` with it so you can see it.

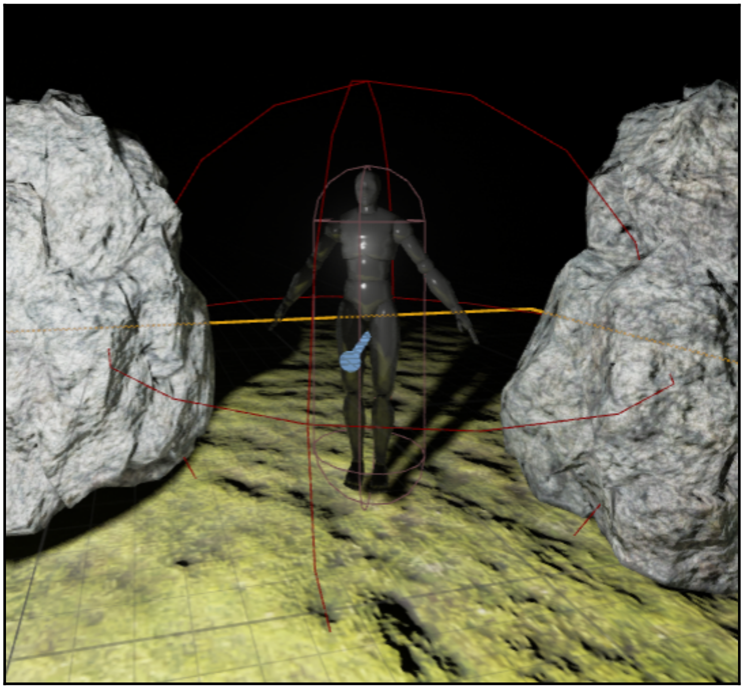
The `Canvas` variable is only available in `DrawHUD()`, so you will have to save `Canvas->SizeX` and `Canvas->SizeY` in class-level variables.

TIP

Refactoring means changing the way code works internally so that it is more organized or easier to read but still has the same apparent result to the user running the program. Refactoring often is a good practice. The reason why refactoring occurs is because nobody knows exactly what the final code should look like when they start writing it.

Triggering an event when the player is near an NPC

To trigger an event near the NPC, we need to set an additional collision detection volume that is a bit wider than the default capsule shape. The additional collision detection volume will be a sphere around each NPC. When the player steps into the NPC sphere, the NPC (shown as follows) reacts and displays a message:



We're going to add the dark red sphere to the NPC so that it can tell when the player is nearby.

Inside your `NPC.h` class file, add `#include "Components/SphereComponent.h"` at the top and the following code:

```
UCLASS() class GOLDENEGG_API ANPC : public ACharacter {
    GENERATED_BODY()

public:
    // The sphere that the player can collide with tob
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
        Collision)
        USphereComponent* ProxSphere;
```

```
    // This is the NPC's message that he has to tell us.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        NPCMessage)
        FString NpcMessage; // The corresponding body of this function is
                            // ANPC::Prox_Implementation, __not__
ANPC::Prox()!
                            // This is a bit weird and not what you'd
expect,
                            // but it happens because this is a
BlueprintNativeEvent
    UFUNCTION(BlueprintNativeEvent, Category = "Collision")
        void Prox(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp,
            int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);
    // You shouldn't need this unless you get a compiler error that it
can't find this function.
    virtual int Prox_Implementation(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp,
        int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);

    // Sets default values for this character's properties
    ANPC(const FObjectInitializer& ObjectInitializer);

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;
};
```

This looks a bit messy, but it is actually not that complicated. Here, we declare an extra bounding sphere volume called `ProxSphere`, which detects when the player is near the NPC.

In the `NPC.cpp` file, we need to add the following code in order to complete the proximity detection:

```
ANPC::ANPC(const FObjectInitializer& ObjectInitializer)
 : Super(ObjectInitializer)
{
```

```
 ProxSphere =
ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this,
 TEXT("Proximity Sphere"));
 ProxSphere->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepWorldTransform);
 ProxSphere->SetSphereRadius(32.0f);
 // Code to make ANPC::Prox() run when this proximity sphere
 // overlaps another actor.
 ProxSphere->OnComponentBeginOverlap.AddDynamic(this, &ANPC::Prox);
 NpcMessage = "Hi, I'm Owen";//default message, can be edited
 // in blueprints
}

// Note! Although this was declared ANPC::Prox() in the header,
// it is now ANPC::Prox_Implementation here.
int ANPC::Prox_Implementation(UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor, UPrimitiveComponent* OtherComp,
 int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // This is where our code will go for what happens
    // when there is an intersection
    return 0;
}
```

Making the NPC display something to the HUD when the player is nearby

When the player is near the NPC sphere-collision volume, display a message to the HUD that alerts the player about what the NPC is saying.

This is the complete implementation of `ANPC::Prox_Implementation`:

```
int ANPC::Prox_Implementation(UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // if the overlapped actor is not the player,
    // you should just simply return from the function
    if( Cast<AAvatar>( OtherActor ) == nullptr ) {
        return -1;
    }
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();
    if( PController )
    {
        AMyHUD * hud = Cast<AMyHUD>( PController->GetHUD() );
```

```
            hud->addMessage( Message( NpcMessage, 5.f, FColor::White ) );
        }
        return 0;
    }
```

Also, make sure you add the following at the top of the file:

```
    #include "Avatar.h"
    #include "MyHud.h"
```

The first thing we do in this function is cast `OtherActor` (the thing that came near the NPC) to `AAvatar`. The cast succeeds (and is not `nullptr`) when `OtherActor` is an `AAvatar` object. We get the HUD object (which happens to be attached to the player controller) and pass a message from the NPC to the HUD. The message is displayed whenever the player is within the red bounding sphere surrounding the NPC:

Jonathan's greeting

Exercises

Try these out for more practice:

1. Add a `UPROPERTY` function name for the NPC's name so that the name of the NPC is editable in blueprints, similar to the message that the NPC has for the player. Show the NPC's name in the output.
2. Add a `UPROPERTY` function (type `UTexture2D*`) for the NPC's face texture. Draw the NPC's face beside its message in the output.
3. Render the player's HP as a bar (filled rectangle).

Solutions

Add the following property to the ANPC class:

```
// This is the NPC's name
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
FString name;
```

Then, in ANPC::Prox_Implementation, change the string passed to the HUD to this:

```
name + FString(": ") + NpcMessage
```

This way, the NPC's name will be attached to the message.

Add the this property to the ANPC class:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
UTexture2D* Face;
```

Then, you can select face icons to be attached to the NPC's face in blueprints.

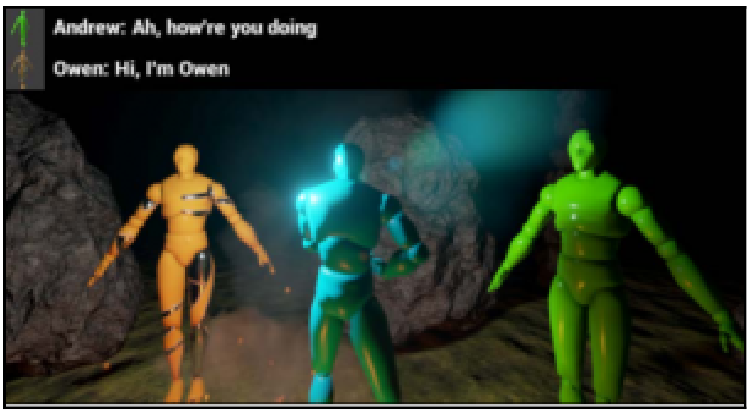
Attach a texture to your struct Message:

```
UTexture2D* tex;
```

To render these icons, you need to add a call to DrawTexture() with the right texture passed in to it:

```
DrawTexture( messages[c].tex, x, y, messageH, messageH, 0, 0, 1, 1
   );
```

Be sure to check whether the texture is valid before you render it. The icons should look similar to what is shown here, at the top of the screen:



This is how a function to draw the player's remaining health in a bar will look:

```
void AMyHUD::DrawHealthbar()
{
    // Draw the healthbar.
    AAvatar *avatar = Cast<AAvatar>(
b        UGameplayStatics::GetPlayerPawn(GetWorld(), 0));
    float barWidth = 200, barHeight = 50, barPad = 12, barMargin = 50;
    float percHp = avatar->Hp / avatar->MaxHp;
    const FVector2D ViewportSize =
FVector2D(GEngine->GameViewport->Viewport->GetSizeXY());
    DrawRect(FLinearColor(0, 0, 0, 1), ViewportSize.X - barWidth -
        barPad - barMargin, ViewportSize.Y - barHeight - barPad -
        barMargin, barWidth + 2 * barPad, barHeight + 2 * barPad);
DrawRect(FLinearColor(1 - percHp, percHp, 0, 1), ViewportSize.X
            - barWidth - barMargin, ViewportSize.Y - barHeight - barMargin,
            barWidth*percHp, barHeight);
}
```

You also need to add `Hp` and `MaxHp` to the Avatar class (you can just set default values for now for testing), and add the following to the top of the file:

```
#include "Kismet/GameplayStatics.h"
#include "Avatar.h"
```

Summary

In this chapter, we went through a lot of material. We showed you how to create a character and display it on the screen, control your character with axis bindings, and create and display NPCs that can post messages to the HUD. It may seem daunting now, but it'll make sense once you get more practice.

In the upcoming chapters, we will develop our game further by adding an inventory system and pickup items, as well as the code and the concept to account for what the player is carrying. Before we do that, though, in the next chapter we will perform an in-depth exploration of some of the UE4 container types.

9
Templates and Commonly-Used Containers

In `Chapter 7`, *Dynamic Memory Allocation*, we spoke about how you would use dynamic memory allocation if you want to create a new array whose size isn't known at compile time. Dynamic memory allocations are of the form `int * array = new int[ number_of_elements ]`.

You also saw that dynamic allocations using the `new[]` keyword require you to call `delete[]` on the array later, otherwise you'd have a memory leak. Having to manage memory this way is hard work.

Is there a way to create an array of a dynamic size and have the memory automatically managed for you by C++? The answer is yes. There are C++ object types (commonly called containers) that handle dynamic memory allocations and deallocations automatically. UE4 provides a couple of container types to store your data in dynamically resizable collections.

There are two different groups of template containers. There is the UE4 family of containers (beginning with `T*`) and the C++ **Standard Template Library** (**STL**) family of containers. There are some differences between the UE4 containers and the C++ STL containers, but the differences are not major. UE4 container sets are written with game performance in mind. C++ STL containers also perform well, and their interfaces are a little more consistent (consistency in an API is something that you'd prefer). Which container set you use is up to you. However, it is recommended that you use the UE4 container set since it guarantees that you won't have cross-platform issues when you try to compile your code.

We will cover the following topics in this chapter:

- Debugging the output in UE4
- Templates and containers
- UE4's TArray
- TSet and TMap
- C++ STL versions of commonly-used containers

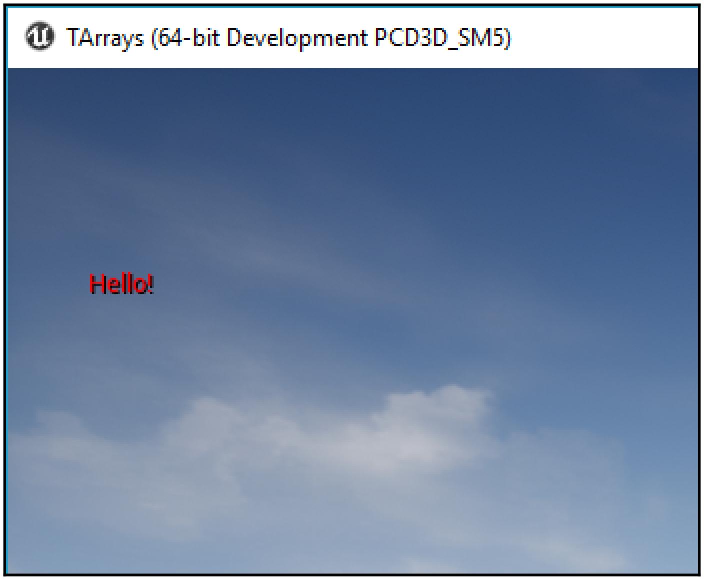
Debugging the output in UE4

All of the code in this chapter (as well as in the later chapters) will require you to work in a UE4 project. For the purpose of testing `TArray`, I created a basic code project called `TArrays`. In the `ATArraysGameMode::ATArraysGameMode` constructor, I am using the debug output feature to print text to the console.

Here's how the code in `TArraysGameMode.cpp` will look:

```
#include "TArraysGameMode.h"
#include "Engine/Engine.h"

ATArraysGameMode::ATArraysGameMode(const FObjectInitializer&
ObjectInitializer) : Super(ObjectInitializer)
{
    if (GEngine)
    {
        GEngine->AddOnScreenDebugMessage(-1, 30.f, FColor::Red,
        TEXT("Hello!"));
    }
}
```

Make sure you also add the function to the `.h` file. If you compile and run this project, you will see the debug text in the top-left corner of your game window when you start the game. You can use debug output to see the internals of your program at any time. Just make sure that the `GEngine` object exists at the time of debugging the output. The output of the preceding code is shown in the following screenshot (note that you may need to run it as a standalone game to see it):



Templates and containers

Templates are a special type of object. A template object lets you specify what type of data it should expect. For example, as you'll see soon, you could run a `TArray<T>` variable. This is an example of a template.

To understand what a `TArray<T>` variable is, you first have to know what the `<T>` option between the angle brackets stands for. The `<T>` option means that the type of data stored in the array is a variable. Do you want an array of `int`? Then create a `TArray<int>` variable. A `TArray` variable of `double`? Create a `TArray<double>` variable.

So in general, wherever `<T>` appears, you can plug in a C++ data type of your choice.

Containers are different structures that are meant for storing objects. Templates are particularly useful for these because they can be used to store many different types of objects. You may want to store numbers with int or float, strings, or different types of game objects. Imagine if you had to write a new class for every type of object you want to store. Fortunately, you don't have to. Templates let one class be flexible enough to handle any objects you want to store in it.

Your first template

Creating templates is an advanced topic, and you could go years without having to create your own (although you'll use the standard ones all the time). But it can be helpful to see what one looks like just to help you understand what's going on behind the scenes.

Imagine you want to create a number template that will let you use an int, float, or another type. You can do something like this:

```
template <class T>
class Number {
    T value;
public:
    Number(T val)
    {
        value = val;
    }

    T getSumWith(T val2);
};

template <class T>
T Number<T>::getSumWith(T val2)
{
    T retval;
    retval = value + val2;
    return retval;
}
```

The first section is the class itself. As you can see, you want to use the type anywhere in the template, you make the class and you will use `T` instead of specifying a particular type. You can also use templates to specify the values sent to functions. In this case, the final section lets you add another number and return the sum.

You can even make things simpler by overloading the + operator so you can add these numbers like you would any standard type. That's through something called operator overloading.

UE4's TArray<T>

TArrays are UE4's Version of a dynamic array, built using templates. Like other dynamic arrays we discussed, you don't have to worry about managing the array size yourself. Let's move on and look at this with an example.

An example that uses TArray<T>

A `TArray<int>` variable is just an array of `int`. A `TArray<Player*>` variable will be an array of `Player*` pointers. An array is dynamically resizable, and elements can be added at the end of the array after its creation.

To create a `TArray<int>` variable, all you have to do is use the normal variable allocation syntax:

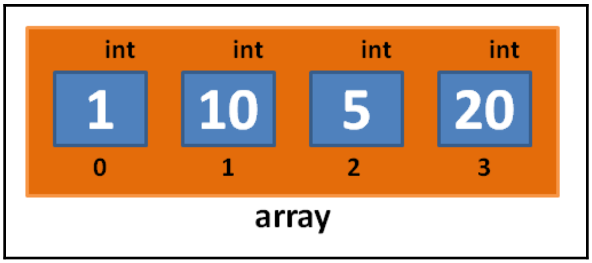
```
TArray<int> array;
```

Changes to the `TArray` variable are done using member functions. There are a couple of member functions that you can use on a `TArray` variable:

The first member function that you need to know about is how you add a value to the array, as shown in the following code:

```
array.Add( 1 );
array.Add( 10 );
array.Add( 5 );
array.Add( 20 );
```

These four lines of code will produce the array value in memory, as shown in the following figure:

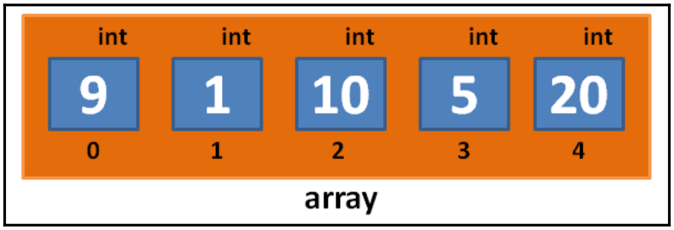


When you call `array.Add( number )`, the new number goes to the end of the array. Since we added the numbers **1**, **10**, **5**, and **20** to the array, in this order, that is the order in which they will go into the array.

If you want to insert a number in the front or middle of the array, this is also possible. All you have to do is use the `array.Insert(value, index)` function, as shown in the following line of code:

```
array.Insert( 9, 0 );
```

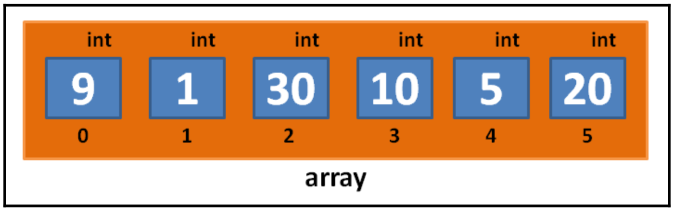
This function will push the number `9` into position `0` of the array (at the front). This means that the rest of the array elements will be offset to the right, as shown in the following figure:



We can insert another element into position `2` of the array using the following line of code:

```
array.Insert( 30, 2 );
```

This function will rearrange the array, as shown in the following diagram:



TIP

If you insert a number into a position in the array that is out of bounds (it doesn't exist), UE4 will crash. So, be careful not to do that. You can use `Add` to add a new item instead.

Iterating a TArray

You can iterate (walk over) the elements of a `TArray` variable in two ways: using integer-based indexing or using an iterator. I will show you both ways here.

The vanilla-for-loop-and-square-brackets notation

Using integers to index the elements of an array is sometimes called a vanilla `for` loop. The elements of the array can be accessed using `array[ index ]`, where `index` is the numerical position of the element in the array:

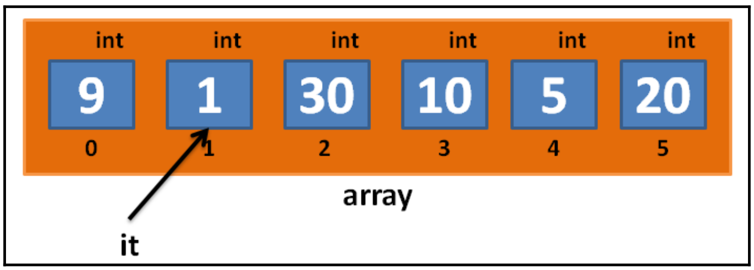
```
for( int index = 0; index < array.Num(); index++ )
{
  // print the array element to the screen using debug message
  GEngine->AddOnScreenDebugMessage( -1, 30.f, FColor::Red,
   FString::FromInt( array[ index ] ) );
}
```

Iterators

You can also use an iterator to walk over the elements of the array one by one, as shown in the following code:

```
for (TArray<int>::TIterator it = array.CreateIterator(); it; ++it)
{
    GEngine->AddOnScreenDebugMessage(-1, 30.f, FColor::Green,
FString::FromInt(*it));
}
```

Iterators are pointers into the array. Iterators can be used to inspect or change values inside the array. An example of an iterator is shown in the following figure:



An iterator is an external object that can look into and inspect the values of an array. Doing `++it` moves the iterator to examine the next element.

An iterator must be suitable for the collection it is walking through. To walk through a `TArray<int>` variable, you need a `TArray<int>::TIterator` type iterator.

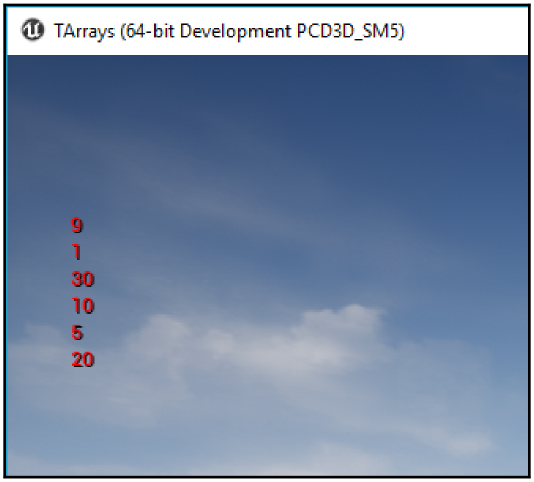
We use `*` to look at the value behind an iterator. In the preceding code, we used `(*it)` to get the integer value from the iterator. This is called dereferencing. To dereference an iterator means to look at its value.

The `++it` operation that happens at the end of each iteration of the `for` loop increments the iterator, moving it on to point to the next element in the list.

Insert the code into the program and try it out now. Here's the example program we have created so far using `TArray` (all in the `ATArraysGameMode::ATArraysGameMode()` constructor):

```
ATArraysGameMode::ATArraysGameMode(const FObjectInitializer&
ObjectInitializer) : Super(ObjectInitializer)
{
    if (GEngine)
    {
        TArray<int> array;
        array.Add(1);
        array.Add(10);
        array.Add(5);
        array.Add(20);
        array.Insert(9, 0);// put a 9 in the front
        array.Insert(30, 2);// put a 30 at index 2
        if (GEngine)
        {
            for (int index = 0; index < array.Num(); index++)
            {
                GEngine->AddOnScreenDebugMessage(index, 30.f, FColor::Red,
                    FString::FromInt(array[index]));
            }
        }
    }
}
```

The output of the preceding code is shown in the following screenshot:



Determining whether an element is in the TArray

Searching our UE4 containers is easy. It is commonly done using the `Find` member function. Using the array we created previously, we can find the index of the value of `10` by typing the following line of code:

```
int index = array.Find( 10 ); // would be index 3 in image above
```

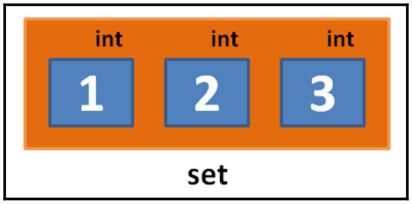
TSet<T>

A `TSet<int>` variable stores a set of integers. A `TSet<FString>` variable stores a set of strings. The main difference between `TSet` and `TArray` is that `TSet` does not allow duplicates; all the elements inside a `TSet` are guaranteed to be unique. A `TArray` variable doesn't mind duplicates of the same elements.

To add numbers to `TSet`, simply call `Add`. Here's an example:

```
TSet<int> set;
set.Add( 1 );
set.Add( 2 );
set.Add( 3 );
set.Add( 1 );// duplicate! won't be added
set.Add( 1 );// duplicate! won't be added
```

This is how `TSet` will look:



Duplicate entries of the same value in `TSet` will not be allowed. Notice how the entries in a `TSet` aren't numbered, as they were in a `TArray`; you can't use square brackets to access an entry in `TSet` arrays.

Iterating a TSet

In order to look into a `TSet` array, you must use an iterator. You can't use the square brackets notation to access the elements of a `TSet`:

```
for( TSet<int>::TIterator it = set.CreateIterator(); it; ++it )
{
  GEngine->AddOnScreenDebugMessage( -1, 30.f, FColor::Red,
   FString::FromInt( *it ) );
}
```

Intersecting TSet arrays

The `TSet` array has two special functions that the `TArray` variable does not. The intersection of two `TSet` arrays is basically the elements they have in common. If we have two `TSet` arrays, such as `X` and `Y`, and we intersect them, the result will be a third, new `TSet` array that contains only the elements common between them. Look at the following example:

```
TSet<int> X;
X.Add( 1 );
X.Add( 2 );
X.Add( 3 );
TSet<int> Y;
Y.Add( 2 );
Y.Add( 4 );
Y.Add( 8 );
TSet<int> common = X.Intersect(Y); // 2
```

The common elements between `X` and `Y` will then just be the element `2`.

Unioning TSet arrays

Mathematically, the union of two sets is when you basically insert all the elements into the same set. Since we are talking about sets here, there won't be any duplicates.

If we take the `X` and `Y` sets from the previous example and create a union, we will get a new set, as follows:

```
TSet<int> uni = X.Union(Y); // 1, 2, 3, 4, 8
```

Finding in TSet arrays

You can determine whether an element is inside a `TSet` or not by using the `Find()` member function on the set. `TSet` will return a pointer to the entry in the `TSet` that matches your query if the element exists in the `TSet`, or it will return `NULL` if the element you're asking for does not exist in the `TSet`.

TMap<T,S>

`TMap<T,S>` creates a table of sorts in the RAM. `TMap` represents a mapping of the keys at the left to the values on the right-hand side. You can visualize `TMap` as a two-column table, with keys in the left column and values in the right column.

A list of items for the player's inventory

For example, say we wanted to create a C++ data structure in order to store a list of items for the player's inventory. On the left-hand side of the table (the keys), we'd have `FString` for the item's name. On the right-hand side (the values), we'd have an `int` for the quantity of that item, as shown in the following table:

Item (key)	Quantity (value)
`apples`	`4`
`donuts`	`12`
`swords`	`1`
`shields`	`2`

To do this in code, we'd simply use the following:

```
TMap<FString, int> items;
items.Add( "apples", 4 );
items.Add( "donuts", 12 );
items.Add( "swords", 1 );
items.Add( "shields", 2 );
```

Once you have created your `TMap`, you can access values inside the `TMap` using square brackets and by passing a key between the brackets. For example, in the `items` map in the preceding code, `items[ "apples" ]` is `4`.

UE4 will crash if you use square brackets to access a key that doesn't exist in the map yet, so be careful! The C++ STL does not crash if you do this.

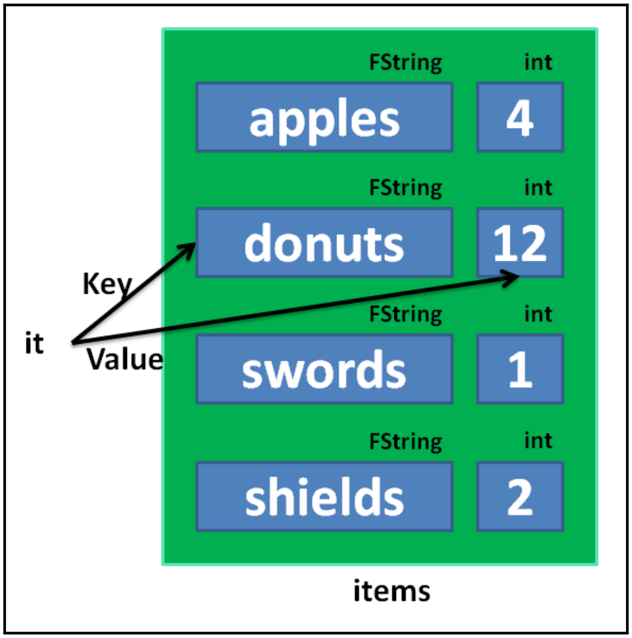
Iterating a TMap

In order to iterate a `TMap`, you use an iterator as well:

```
for( TMap<FString, int>::TIterator it = items.CreateIterator(); it; ++it )
{
  GEngine->AddOnScreenDebugMessage( -1, 30.f, FColor::Red,
  it->Key + FString(": ") + FString::FromInt( it->Value ) );
}
```

`TMap` iterators are slightly different from `TArray` or `TSet` iterators. A `TMap` iterator contains both a `Key` and a `Value`. We can access the key with `it->Key` and the value inside the `TMap` with `it->Value`.

Here's an example:

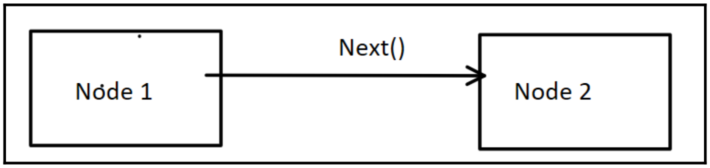


TLinkedList/TDoubleLinkedList

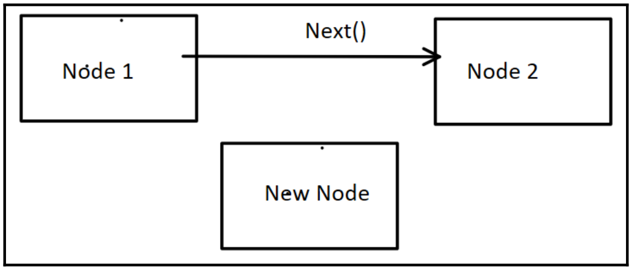
When you work with a TArray, each item has an index in numerical order, and array data is generally stored the same way, so each entry is right next to the one before it in memory too. But what if you need to put a new item somewhere in the middle (for example, if the array is filled with strings in alphabetical order)?

Since the items are next to each other, the one next to it will have to be moved over to make room. But to do that, the one next to that will also have to be moved over. This will continue until the end of the array, when it finally gets to memory it can use without moving something else. As you might imagine, this could get very slow, especially if you're doing it a lot.

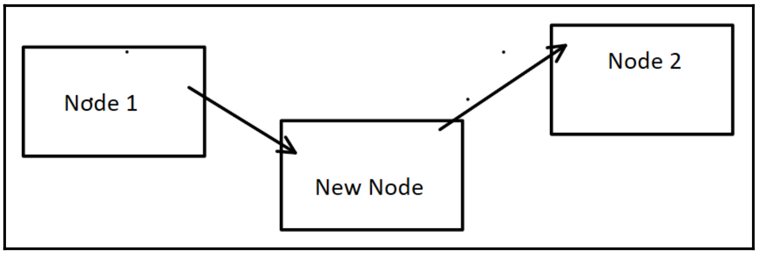
This is where linked lists come in. A linked list doesn't have any indices. A linked list has nodes that contain the items and give you access to the first node on the list. That node has a pointer to the next node on the list, which you can get by calling `Next()`. Then, you can call `Next()` on that one to get the one after it. It looks something like this:



As you might guess, this could get slow if you're looking for an item at the end of the list. But at the same time, you might not be searching the list that often, and might instead be adding new items somewhere in the middle. Adding an item in the middle is a lot faster. Say you're trying to insert a new node between **Node 1** and **Node 2**, like this:



There's no need to move things around in memory to make room this time. Instead, to insert an item after another one, get the node that `Next()` points to from **Node 1** (**Node 2**). Set the new node to point to that one (**Node 2**). Then, set Node 1 to point to the new node. It should now look something like this:



And you're done!

So what if you are going to be spending more time looking for items toward the end of the list? That is where `TDoubleLinkedList` comes in handy. Doubly-linked lists can give you either the first node in the list or the last node in the list. Each node also has pointers to both the next node and the previous node. You can access these using `GetNextLink()` and `GetPrevLink()`. So, you have the choice to go forward or backward through the list, or even to do both and meet in the middle.

Now, you may ask yourself, *"Why does it matter when I can just use TArray and not worry about what it's doing behind the scenes?"* For one thing, professional game programmers always have to worry about speed. Every advance in computers and game consoles is matched by more and better graphics, and other advances that slow things right back down again. So, optimizing speed is always important.

Plus, there's the other practical reason: I can tell you from experience that there are people in this industry who will turn you down in job interviews if you don't use linked lists. Programmers all have their own preferred ways of doing things, so you should always be familiar with anything that might come up.

C++ STL versions of commonly-used containers

Now, we'll cover the C++ STL versions of a few containers. STL is the standard template library, which is shipped with most C++ compilers. The reason why I want to cover these STL versions is that they behave somewhat differently than the UE4 versions of the same containers. In some ways, their behavior is very good, but game programmers often complain of STL having performance issues. In particular, I want to cover STL's `set` and `map` containers, but I will also cover the commonly-used `vector`.

> If you like STL's interface but want better performance, there is a well-known reimplementation of the STL library by Electronic Arts called EASTL, which you can use. It provides the same functionality as STL but is implemented with better performance (basically by doing things such as eliminating bounds checking). It is available on GitHub at `https://github.com/paulhodge/EASTL`.

The C++ STL set

A C++ set is a bunch of items that are unique and sorted. The good feature about the STL `set` is that it keeps the set elements sorted. A quick and dirty way to sort a bunch of values is actually to just shove them into the same `set`. The `set` will take care of the sorting for you.

We can return to a simple C++ console application for the use of sets. To use the C++ STL set, you need to include `<set>`, as shown here:

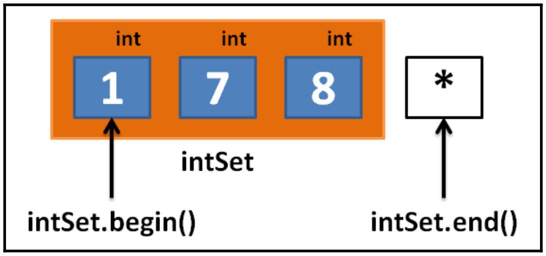
```
#include <iostream>
#include <set>
using namespace std;

int main()
{
  set<int> intSet;
  intSet.insert( 7 );
  intSet.insert( 7 );
  intSet.insert( 8 );
  intSet.insert( 1 );

  for( set<int>::iterator it = intSet.begin(); it != intSet.end();
   ++it )
  {
    cout << *it << endl;
  }
}
```

The following is the output of the preceding code:

```
1
7
8
```

The duplicate `7` is filtered out, and the elements are kept in increasing order inside the `set`. The way we iterate over the elements of an STL container is similar to UE4's `TSet` array. The `intSet.begin()` function returns an iterator that points to the head of `intSet`.

The condition to stop iterating is when it becomes `intSet.end()`. `intSet.end()` is actually one position past the end of the `set`, as shown in the following figure:



Finding an element in a <set>

To find an element inside an STL `set`, we can use the `find()` member function. If the item we're looking for turns up in the `set`, we get an iterator that points to the element we were searching for. If the item that we were looking for is not in the `set`, we get back `set.end()` instead, as shown here:

```
set<int>::iterator it = intSet.find( 7 );
if( it != intSet.end() )
{
  //  7  was inside intSet, and *it has its value
  cout << "Found " << *it << endl;
}
```

Exercise

Ask the user for a set of three unique names. Take each name in, one by one, and then print them in a sorted order. If the user repeats a name, ask them for another one until you get to three.

Solution

The solution for the preceding exercise can be found using the following code:

```
#include <iostream>
#include <string>
#include <set>
using namespace std;
int main()
{
  set<string> names;
  // so long as we don't have 3 names yet, keep looping,
  while( names.size() < 3 )
```

```
  {
    cout << names.size() << " names so far. Enter a name" << endl;
    string name;
    cin >> name;
    names.insert( name ); // won't insert if already there,
  }
  // now print the names. the set will have kept order
  for( set<string>::iterator it = names.begin(); it !=
   names.end(); ++it )
  {
    cout << *it << endl;
  }
}
```

The C++ STL map

The C++ STL `map` object is a lot like UE4's `TMap` object. The one thing it does that `TMap` does not is maintain a sorted order inside the map as well. Sorting introduces an additional cost, but if you want your map to be sorted, opting for the STL Version might be a good choice.

To use the C++ STL `map` object, we include `<map>`. In the following example program, we populate a map of items with some key-value pairs:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
  map<string, int> items;
  items.insert( make_pair( "apple", 12 ) );
  items.insert( make_pair( "orange", 1 ) );
  items.insert( make_pair( "banana", 3 ) );
  // can also use square brackets to insert into an STL map
  items[ "kiwis" ] = 44;

  for( map<string, int>::iterator it = items.begin(); it !=
   items.end(); ++it )
  {
    cout << "items[ " << it->first << " ] = " << it->second <<
     endl;
  }
}
```

This is the output of the preceding program:

```
items[ apple ] = 12
items[ banana ] = 3
items[ kiwis ] = 44
items[ orange ] = 1
```

Notice how the iterator's syntax for an STL map is slightly different than that of `TMap`; we access the key using `it->first` and the value using `it->second`.

Notice how C++ STL also offers a bit of syntactic sugar over `TMap`; you can use square brackets to insert into the C++ STL `map`. You cannot use square brackets to insert into a `TMap`.

Finding an element in a <map>

You can search a map for a `<key, value>` pair using the STL map's `find` member function. You generally search by `key` and it'll give you the value for that `key`.

Exercise

Ask the user to enter five items and their quantities into an empty `map`. Print the results in sorted order (that is, alphabetically or lowest to highest in the case of numbers).

Solution

The solution for the preceding exercise uses the following code:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
  map<string, int> items;
  cout << "Enter 5 items, and their quantities" << endl;
  while( items.size() < 5 )
  {
    cout << "Enter item" << endl;
    string item;
    cin >> item;
    cout << "Enter quantity" << endl;
    int qty;
```

```
      cin >> qty;
      items[ item ] = qty; // save in map, square brackets
      // notation
    }

    for( map<string, int>::iterator it = items.begin(); it !=
     items.end(); ++it )
    {
      cout << "items[ " << it->first << " ] = " << it->second <<
       endl;
    }
  }
```

In this solution code, we start by creating `map<string, int> items` to store all the items we're going to take in. Ask the user for an item and a quantity; then, we save the `item` in the `items` map using the square brackets notation.

C++ STL Vector

`Vector` is the STL equivalent of `TArray`. It's basically an array that manages everything behind the scenes, the same way `TArray` does. You may not need to use it when working in UE4, but it's good to know in case someone else uses it in a project.

Summary

UE4's containers and the C++ STL family of containers are both excellent for storing game data. Often, a programming problem can be simplified a lot by selecting the right type of data container.

In the next chapter, we will actually start to program the beginning of our game by keeping track of what the player is carrying and storing that information in a `TMap` object.

10
Inventory System and Pickup Items

We want our player to be able to pick up items from the game world. In this chapter, we will code and design a backpack for our player to store items. We will display what the player is carrying in the pack when the user presses the *I* key.

As a data representation, we can use the `TMap<FString, int>` items covered in the previous chapter to store our items. When the player picks up an item, we add it to the map. If the item is already in the map, we just increase its value by the quantity of the new items picked up.

We will be covering the following topics in this chapter:

- Declaring the backpack
- The PickupItem base class
- Drawing the player inventory

Declaring the backpack

We can represent the player's backpack as a simple `TMap<FString, int>` item. To allow our player to gather items from the world, open the `Avatar.h` file and add the following `TMap` declaration:

```
class APickupItem; //  forward declare the APickupItem class,
                   // since it will be "mentioned" in a member
                    function decl below
UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
  GENERATED_BODY()
```

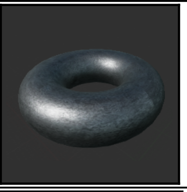
```
public:
  // A map for the player's backpack
  TMap<FString, int> Backpack;

  // The icons for the items in the backpack, lookup by string
  TMap<FString, UTexture2D*> Icons;

  // A flag alerting us the UI is showing
  bool inventoryShowing;
  // member function for letting the avatar have an item
  void Pickup( APickupItem *item );
  // ... rest of Avatar.h same as before
};
```

Forward declaration

Before the `AAvatar` class, notice that we have a `class APickupItem` forward declaration. Forward declarations are needed in a code file when a class is mentioned (such as the `APickupItem::Pickup( APickupItem *item );` function prototype), but there is no code in the file actually using an object of that type inside the file. Since the `Avatar.h` header file does not contain executable code that uses an object of the type `APickupItem`, a forward declaration is what we need. While it may seem easier to include a .h file. sometimes it's better to avoid that, or you may get circular dependencies (two classes that each try including the other one can cause problems).

The absence of a forward declaration will give a compiler error, since the compiler won't have heard of `class APickupItem` before compiling the code in `class AAvatar`. The compiler error will come at the declaration of the `APickupItem::Pickup( APickupItem *item );` function prototype declaration.

We declared two `TMap` objects inside the `AAvatar` class. This is how the objects will look, as shown in the following table:

`FString` (name)	`int` (quantity)	`UTexture2D*` (im)
`GoldenEgg`	`2`	
`MetalDonut`	`1`	
`Cow`	`2`	

In the `TMap` backpack, we store the `FString` variable of the item that the player is holding. In the `Icons` map, we store a single reference to the image of the item the player is holding.

At render time, we can use the two maps working together to look up both the quantity of an item that the player has (in his `Backpack` mapping), and the texture asset reference of that item (in the `Icons` map). The following screenshot shows how the rendering of the HUD will look:

Note that we can also use an array of `struct` with an `FString` variable and `UTexture2D*` in it instead of using two maps.

For example, we can keep `TArray<Item> Backpack;` with a `struct` variable, as shown in the following code:

```
struct Item
{
  FString name;
  int qty;
  UTexture2D*   tex;
};
```

Then, as we pick up items, they will be added to the linear array. However, counting the number of each item we have in the backpack will require constant re-evaluation by iterating through the array of items each time we want to see the count. For example, to see how many hairbrushes you have, you will need to make a pass through the whole array. This is not as efficient as using a map.

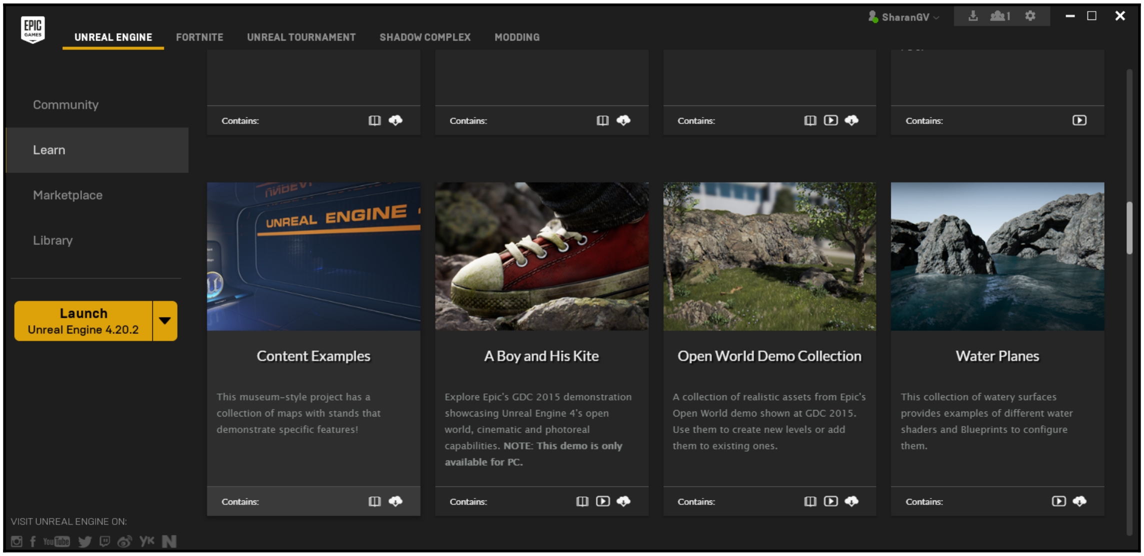
Importing assets

You might have noticed the **Cow** asset in the preceding screenshot, which is not a part of the standard set of assets that UE4 provides in a new project. In order to use the **Cow** asset, you need to import the cow from the **Content Examples** project. There is a standard importing procedure that UE4 uses.

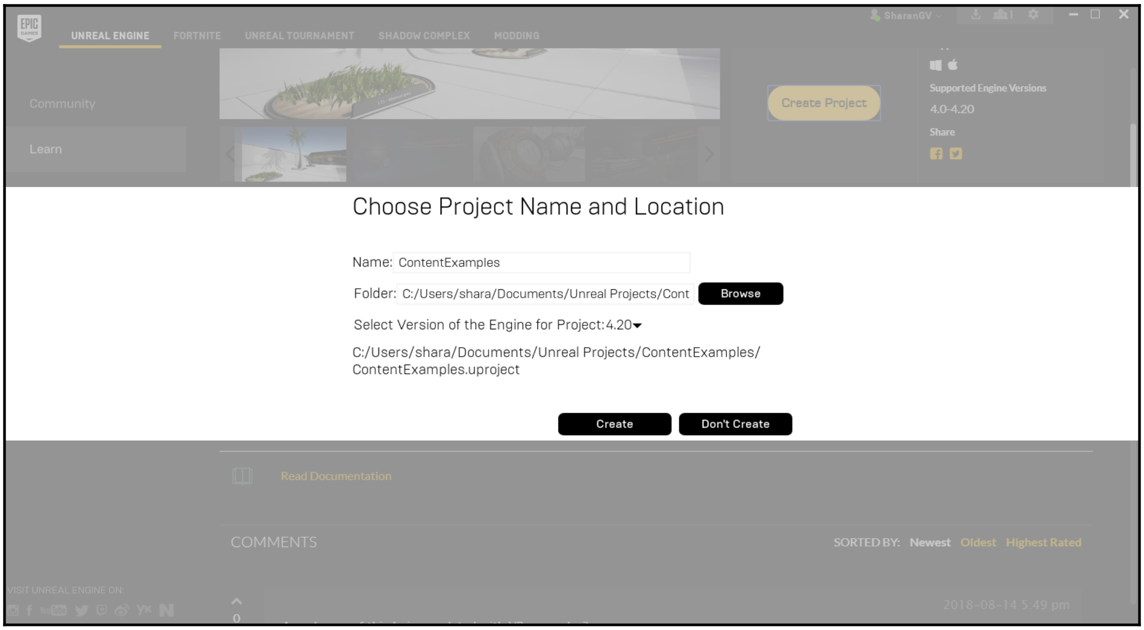
In the following screenshot, I have outlined the procedure for importing the **Cow** asset. Other assets will be imported from other projects in UE4 using the same method.

Perform the following steps to import the **Cow** asset:

1. Download and open UE4's **Content Examples** project. Find it under **Learn** in the Epic Game Launcher, shown as follows:



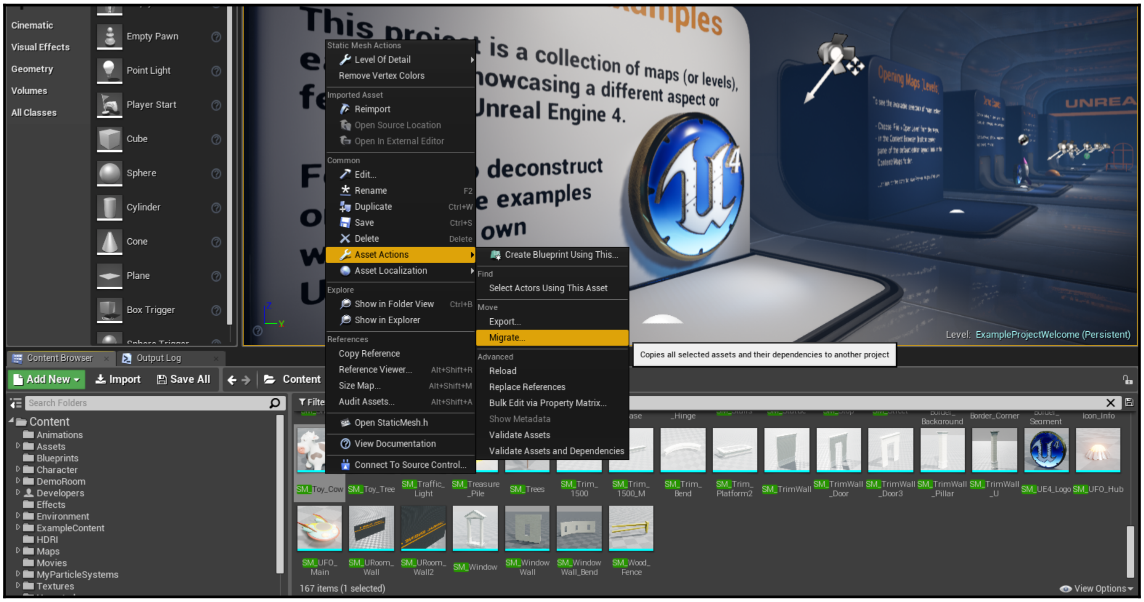
2. After you have downloaded **Content Examples**, open it and click on **Create Project**:



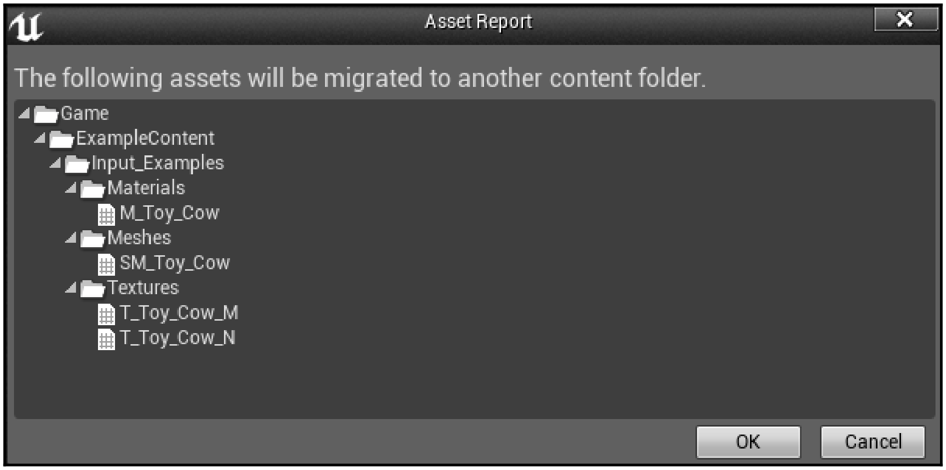
3. Next, name the folder in which you will put your `ContentExamples` and click on **Create**.
4. Open your `ContentExamples` project from the library. Browse the assets available in the project until you find one that you like. Searching for `SM_` will help since all static meshes usually begin with `SM_` by convention:

Assets available in the project

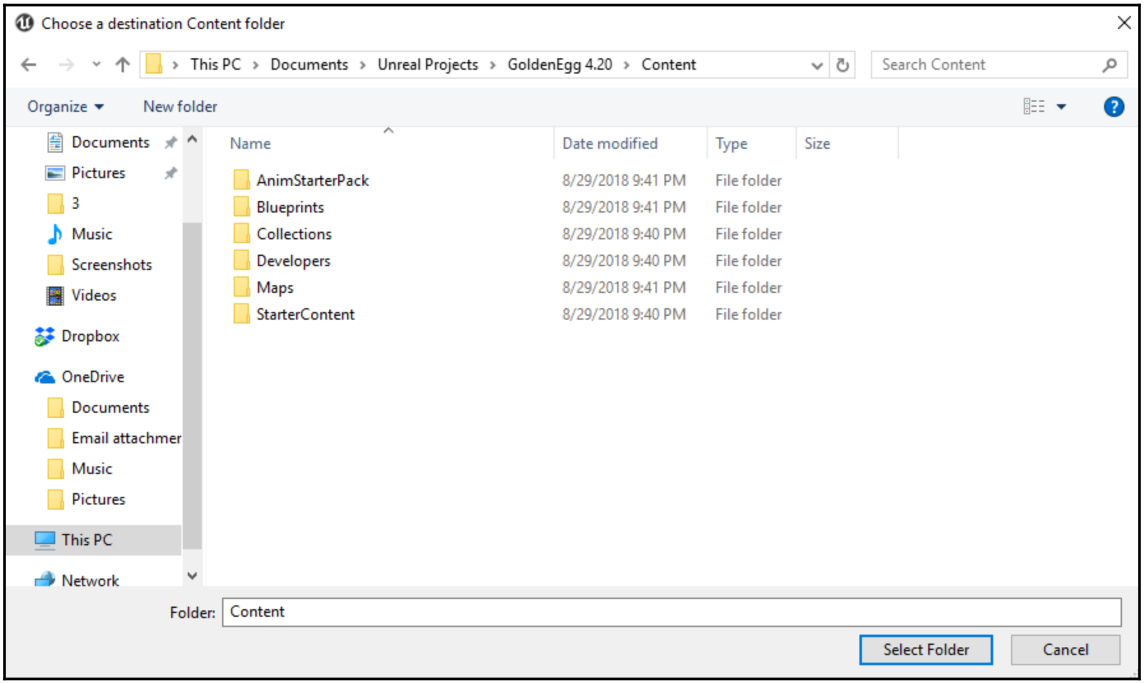
5. When you find an asset that you like, import it into your project by right-clicking on the asset and then clicking on **Asset Actions > Migrate...**:



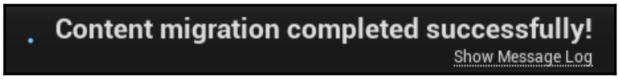
6. Click on **OK** in the **Asset Report** dialog:



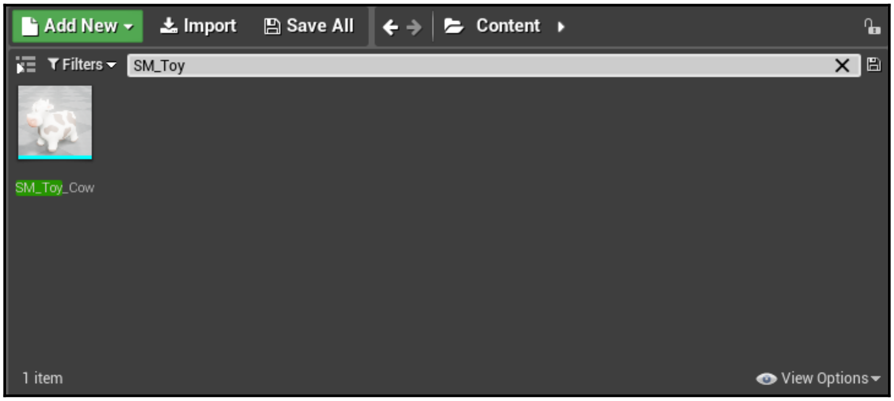
7. Select the **Content** folder from your project that you want to add the **SM_Toy_Cow** file to. We will add it to `/Documents/Unreal Projects/GoldenEgg/Content`, as shown in the following screenshot:



8. If the import was completed successfully, you will see the following message:



9. Once you import your asset, you will see it show up in your asset browser inside your project:

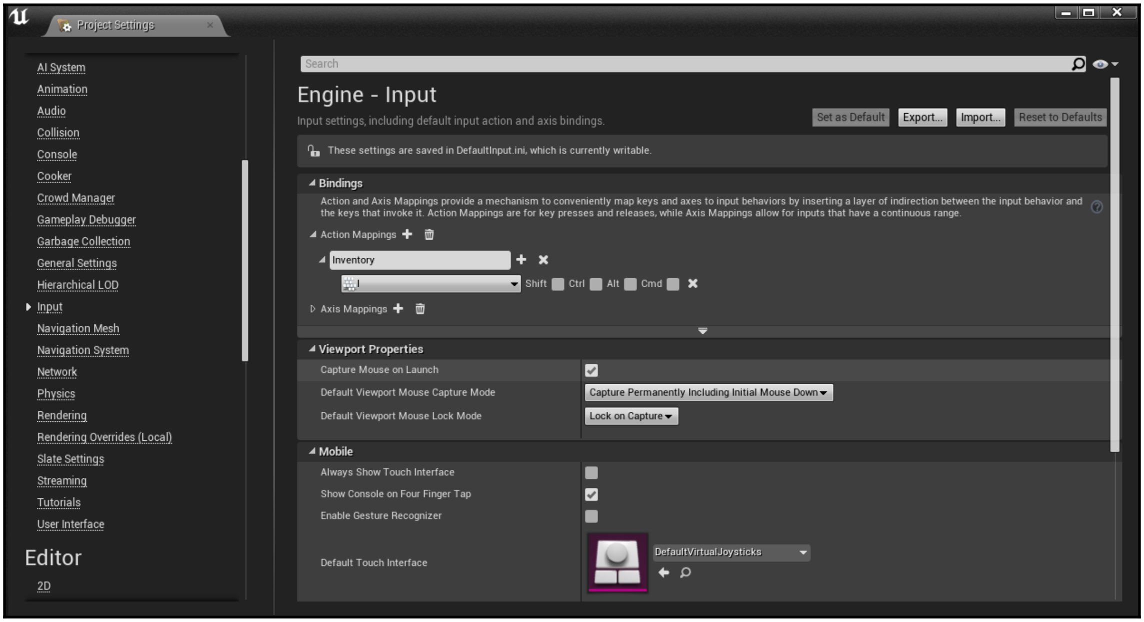


You can then use the asset inside your project normally.

Attaching an action mapping to a key

We need to attach a key to activate the display of the player's inventory. Inside the UE4 editor, follow these steps:

1. Add an **Action Mappings +** called `Inventory`
2. Assign it to the keyboard key *I*, as shown:



3. Next, in the `Avatar.h` file, add a member function to be run when the player's inventory needs to be displayed:

```
void ToggleInventory();
```

4. In the `Avatar.cpp` file, implement the `ToggleInventory()` function, as shown in the following code:

```
void AAvatar::ToggleInventory()
{
  if( GEngine )
  {
    GEngine->AddOnScreenDebugMessage( -1, 5.f, FColor::Red,
     "Showing inventory..." );
  }
}
```

5. Then, connect the `"Inventory"` action to `AAvatar::ToggleInventory()` in `SetupPlayerInputComponent()`:

```
void AAvatar::SetupPlayerInputComponent(class UInputComponent*
   InputComponent)
{
 Super::SetupPlayerInputComponent(PlayerInputComponent);
```

```
        check(PlayerInputComponent);
        PlayerInputComponent->BindAction("Inventory", IE_Pressed, this,
            &AAvatar::ToggleInventory);
      // rest of SetupPlayerInputComponent same as before
    }
```

The PickupItem base class

We need to define how a pickup item looks in code. Each pickup item will be derived from a common base class. Let's construct the base class for a `PickupItem` class now.

The `PickupItem` base class should inherit from the `AActor` class. Similar to how we created multiple NPC blueprints from the base NPC class, we can create multiple `PickupItem` blueprints from a single `PickupItem` base class, as shown in the following screenshot:

The text in this screenshot is not important. this image gives you an idea of how to create multiple `PickupItem` blueprints from a single `PickupItem` base class

Once you have created the `PickupItem` class, open its code in Visual Studio.

The `APickupItem` class will need quite a few members, as follows:

- An `FString` variable for the name of the item being picked up
- An `int32` variable for the quantity of the item being picked up
- A `USphereComponent` variable for the sphere that you will collide with for the item to be picked up
- A `UStaticMeshComponent` variable to hold the actual `Mesh`
- A `UTexture2D` variable for the icon that represents the item
- A pointer for the HUD (which we will initialize later)

This is how the code in `PickupItem.h` looks:

```
// Fill out your copyright notice in the Description page of Project
Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Components/SphereComponent.h"
#include "Components/StaticMeshComponent.h"
#include "PickupItem.generated.h"

UCLASS()
class GOLDENEGG_API APickupItem : public AActor
{
    GENERATED_BODY()
public:
    // Sets default values for this actor's properties
    APickupItem(const FObjectInitializer& ObjectInitializer);

    // The name of the item you are getting
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
        FString Name;

    // How much you are getting
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
        int32 Quantity;

    // the sphere you collide with to pick item up
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Item)
        USphereComponent* ProxSphere;

    // The mesh of the item
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Item)
        UStaticMeshComponent* Mesh;
    // The icon that represents the object in UI/canvas
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
        UTexture2D* Icon;
    // When something comes inside ProxSphere, this function runs
    UFUNCTION(BlueprintNativeEvent, Category = Collision)
        void Prox(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp,
            int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);
        virtual int Prox_Implementation(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp,
        int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
```

```
SweepResult);

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;
};
```

The point of all these `UPROPERTY()` declarations is to make `APickupItem` completely configurable by blueprints. For example, the items in the **Pickup** category will be displayed as follows in the blueprints editor:

In the `PickupItem.cpp` file, we complete the constructor for the `APickupItem` class, as shown in the following code:

```
APickupItem::APickupItem(const FObjectInitializer& ObjectInitializer)
    : Super(ObjectInitializer)
{
    Name = "UNKNOWN ITEM";
    Quantity = 0;

    // initialize the unreal objects
    ProxSphere =
ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this,
        TEXT("ProxSphere"));
    Mesh =
ObjectInitializer.CreateDefaultSubobject<UStaticMeshComponent>(this,
            TEXT("Mesh"));

    // make the root object the Mesh
    RootComponent = Mesh;
    Mesh->SetSimulatePhysics(true);
```

```
    // Code to make APickupItem::Prox() run when this
    // object's proximity sphere overlaps another actor.
    ProxSphere->OnComponentBeginOverlap.AddDynamic(this,
&APickupItem::Prox);
    ProxSphere->AttachToComponent(Mesh,
FAttachmentTransformRules::KeepWorldTransform); // very important!
}
```

In the first two lines, we perform an initialization of `Name` and `Quantity` to values that should stand out to the game designer as being uninitialized. We used block capitals so that the designer can clearly see that the variable has never been initialized before.

We then initialize the `ProxSphere` and `Mesh` components using `ObjectInitializer.CreateDefaultSubobject`. The freshly initialized objects might have some of their default values initialized, but `Mesh` will start out empty. You will have to load the actual mesh later, inside blueprints.

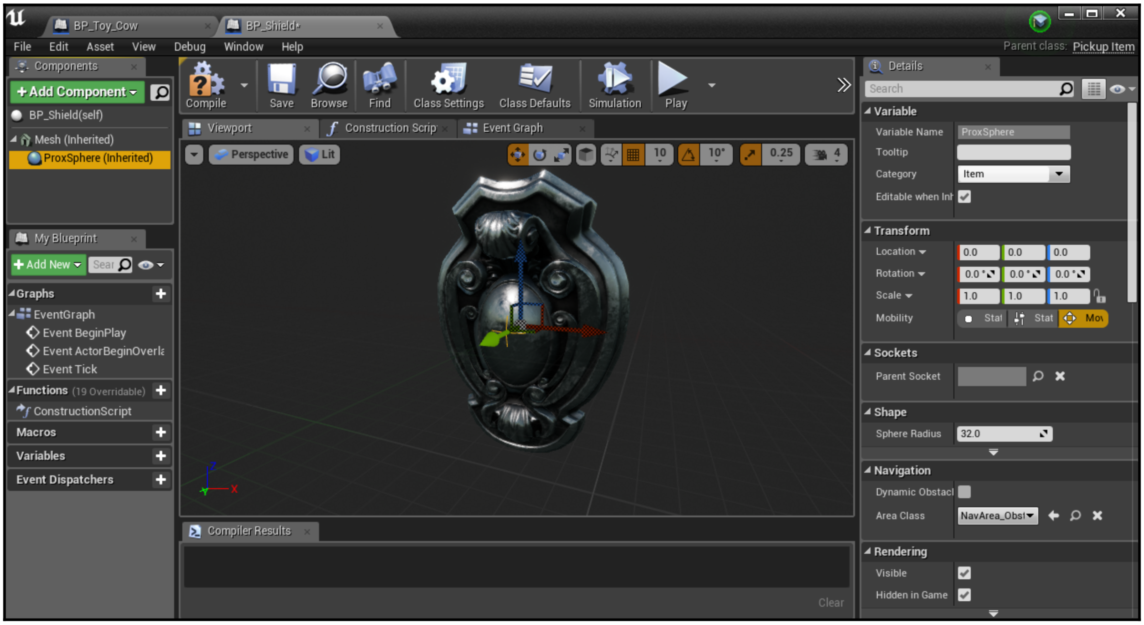
For the mesh, we set it to simulate realistic physics so that pickup items will bounce and roll around if they are dropped or moved. Pay special attention to the line `ProxSphere->AttachToComponent(Mesh, FAttachmentTransformRules::KeepWorldTransform);`. This line tells you to make sure the pickup item's `ProxSphere` component is attached to the `Mesh` root component. This means that when the mesh moves in the level, `ProxSphere` follows. If you forget this step (or if you did it the other way around), then `ProxSphere` will not follow the mesh when it bounces.

The root component

In the preceding code, we assigned `RootComponent` of `APickupItem` to the `Mesh` object. The `RootComponent` member is a part of the `AActor` base class, so every `AActor` and its derivatives have a root component. The root component is basically meant to be the core of the object, and also defines how you collide with the object. The `RootComponent` object is defined in the `Actor.h` file, as shown in the following code:

```
/** Collision primitive that defines the transform (location, rotation,
scale) of this Actor. */
    UPROPERTY(BlueprintGetter=K2_GetRootComponent,
Category="Utilities|Transformation")
    USceneComponent* RootComponent;
```

So, the UE4 creators intended `RootComponent` to always be a reference to the collision primitive. Sometimes the collision primitive can be capsule shaped—other times it can be spherical or even box-shaped, or it can be arbitrarily shaped, as in our case, with the mesh. It's rare that a character should have a box-shaped root component, however, because the corners of the box can get caught on walls. Round shapes are usually preferred. The `RootComponent` property shows up in the blueprints, where you can see and manipulate it:



You can edit the ProxSphere root component from its blueprints once you create a blueprint based on the PickupItem class

Finally, the `Prox_Implementation` function gets implemented, as follows:

```
int APickupItem::Prox_Implementation(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // if the overlapped actor is NOT the player,
    // you simply should return
    if (Cast<AAvatar>(OtherActor) == nullptr)
    {
        return -1;
    }

    // Get a reference to the player avatar, to give him
```

```
    // the item
    AAvatar *avatar =
Cast<AAvatar>(UGameplayStatics::GetPlayerPawn(GetWorld(), 0));

    // Let the player pick up item
    // Notice use of keyword this!
    // That is how _this_ Pickup can refer to itself.
    avatar->Pickup(this);

    // Get a reference to the controller
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();

    // Get a reference to the HUD from the controller
    AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());
    hud->addMessage(Message(Icon, FString("Picked up ") +
FString::FromInt(Quantity) + FString(" ") + Name, 5.f, FColor::White)
);

    Destroy();

    return 0;
}
```

Also, make sure you add the following at the top of the file:

```
#include "Avatar.h"
#include "MyHUD.h"
#include "Kismet/GameplayStatics.h"
```

A couple of tips here that are pretty important: first, we have to access a couple of *globals* to get the objects we need. There are three main objects we'll be accessing through these functions that manipulate the HUD:

- The controller (`APlayerController`)
- The HUD (`AMyHUD`)
- The player himself (`AAvatar`)

There is only one of each of these three types of objects in the game instance. UE4 has made finding them easy.

Also, for this to compile you also need to add another constructor to the `Message` struct in `MyHud.h`. You need one that lets you pass in the image like this:

```
Message(UTexture2D* img, FString iMessage, float iTime, FColor iColor)
    {
        tex = img;
        message = iMessage;
        time = iTime;
        color = iColor;
    }
```

To compile, you will also need to add another variable to the struct, `UTexture2D* tex;`. You also need to implement the Pickup function in Avatar.

Getting the avatar

The `player` class object can be found at any time from any place in the code by simply calling the following code:

```
AAvatar *avatar = Cast<AAvatar>(
  UGameplayStatics::GetPlayerPawn( GetWorld(), 0 ) );
```

We then pass the player the item by calling the `AAvatar::Pickup()` function defined earlier.

Because the `PlayerPawn` object is really an `AAvatar` instance, we cast the result to the `AAvatar` class, using the `Cast<AAvatar>` command. The `UGameplayStatics` family of functions are accessible anywhere in your code as they are global functions.

Getting the player controller

Retrieving the player controller can be done from a global function as well:

```
APlayerController* PController =
  GetWorld()->GetFirstPlayerController();
```

The `GetWorld()` function is actually defined in the `UObject` base class. Since all UE4 objects derive from `UObject`, any object in the game actually has access to the `world` object.

Getting the HUD

Although this organization might seem strange at first, the HUD is actually attached to the player's controller. You can retrieve the HUD as follows:

```
AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
```

We cast the HUD object since we previously set the HUD to be an `AMyHUD` instance in blueprints. Since we will be using the HUD often, we can actually store a permanent pointer to the HUD inside our `APickupItem` class. We will discuss this point later.

Next, we implement `AAvatar::Pickup`, which adds an object of the type `APickupItem` to the Avatar's backpack:

```
void AAvatar::Pickup(APickupItem *item)
{
    if (Backpack.Find(item->Name))
    {
        // the item was already in the pack.. increase qty of it
        Backpack[item->Name] += item->Quantity;
    }
    else
    {
        // the item wasn't in the pack before, add it in now
        Backpack.Add(item->Name, item->Quantity);
        // record ref to the tex the first time it is picked up
        Icons.Add(item->Name, item->Icon);
    }
}
```

Also, make sure you add `#include "PickupItem.h"` at the top of the file.

In the preceding code, we check whether the pickup item that the player just got is already in his pack. If it is, we increase its quantity. If it is not in his pack, we add it to both his pack and the `Icons` mapping.

To add the pickup items to the pack, use the following line of code:

```
avatar->Pickup( this );
```

`APickupItem::Prox_Implementation` is the way this member function will get called.

Now, we need to display the contents of our backpack in the HUD when the player presses *I*.

Drawing the player inventory

An inventory screen in a game such as *Diablo* features a pop-up window, with the icons of the items you've picked up in the past arranged in a grid. We can achieve this type of behavior in UE4.

There are a number of approaches to drawing a UI in UE4. The most basic way is to simply use `HUD::DrawTexture()` calls. Another way is to use Slate. Another way still is to use the newest UE4 UI functionality: **Unreal Motion Graphics** (**UMG**) Designer.

Slate uses a declarative syntax to lay out UI elements in C++. Slate is best suited for menus and the like. UMG has been around since UE 4.5 and uses a heavily blueprint-based workflow. Since our focus here is on exercises that use C++ code, we will stick to a `HUD::DrawTexture()` implementation, but we will go over UMG in a later chapter. This means that we will have to manage all the data that deals with the inventory in our code.

Using HUD::DrawTexture()

`HUD::DrawTexture()` is what we will use to draw the inventory to the screen at this point. We will achieve this in two steps:

1. We push the contents of our inventory to the HUD when the user presses the *I* key.
2. Then, we render the icons into the HUD in a grid-like fashion.

   To keep all the information about how a widget can be rendered, we declare a simple structure to keep the information concerning what icon it uses, its current position, and current size.

   This is how the `Icon` and `Widget` structures look:

```
struct Icon
{
  FString name;
  UTexture2D* tex;
  Icon(){ name = "UNKNOWN ICON"; tex = 0; }
  Icon( FString& iName, UTexture2D* iTex )
  {
    name = iName;
    tex = iTex;
  }
};
```

```
struct Widget
{
  Icon icon;
  FVector2D pos, size;
  Widget(Icon iicon)
  {
    icon = iicon;
  }
  float left(){ return pos.X; }
  float right(){ return pos.X + size.X; }
  float top(){ return pos.Y; }
  float bottom(){ return pos.Y + size.Y; }
};
```

You can add these structure declarations to the top of `MyHUD.h`, or you can add them to a separate file and include that file everywhere those structures are used.

Notice the four member functions on the `Widget` structure to get to the `left()`, `right()`, `top()`, and `bottom()` functions of the widget. We will use these later to determine whether a click point is inside the box.

3. Next, we declare the function that will render the widgets out on the screen in the `AMyHUD` class. First, in `MyHud.h`, add an array to hold widgets and a vector to hold the screen dimensions:

```
// New! An array of widgets for display
TArray<Widget> widgets;
//Hold screen dimensions
FVector2D dims;
```

4. Also, add the line `void DrawWidgets();`. Then, add this to `MyHud.cpp`:

```
void AMyHUD::DrawWidgets()
{
    for (int c = 0; c < widgets.Num(); c++)
    {
        DrawTexture(widgets[c].icon.tex, widgets[c].pos.X,
            widgets[c].pos.Y, widgets[c].size.X, widgets[c].size.Y,
0, 0,
            1, 1);    DrawText(widgets[c].icon.name,
FLinearColor::Yellow,
                widgets[c].pos.X, widgets[c].pos.Y, hudFont, .6f,
false);
    }
}
```

5. A call to the `DrawWidgets()` function should be added to the `DrawHUD()` function, and you might want to move the current message handling code into a separate `DrawMessages` function so you can then get this (or just leave the original code there):

```
void AMyHUD::DrawHUD()
{
    Super::DrawHUD();
    // dims only exist here in stock variable Canvas
    // Update them so use in addWidget()
    const FVector2D ViewportSize =
FVector2D(GEngine->GameViewport->Viewport->GetSizeXY());
    dims.X = ViewportSize.X;
    dims.Y = ViewportSize.Y;
    DrawMessages();
    DrawWidgets();
}
```

6. Next, we will fill the `ToggleInventory()` function. This is the function that runs when the user presses *I*:

```
void AAvatar::ToggleInventory()
{
    // Get the controller & hud
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());

    // If inventory is displayed, undisplay it.
    if (inventoryShowing)
    {
        hud->clearWidgets();
        inventoryShowing = false;
        PController->bShowMouseCursor = false;
        return;
    }

    // Otherwise, display the player's inventory
    inventoryShowing = true;
    PController->bShowMouseCursor = true;
    for (TMap<FString, int>::TIterator it =
        Backpack.CreateIterator(); it; ++it)
    {
        // Combine string name of the item, with qty eg Cow x 5
        FString fs = it->Key + FString::Printf(TEXT(" x %d"),
it->Value);
        UTexture2D* tex;
```

```
            if (Icons.Find(it->Key))
            {
                tex = Icons[it->Key];
                hud->addWidget(Widget(Icon(fs, tex)));
            }
        }
    }
```

7. For the preceding code to compile, we need to add two functions to `AMyHUD`:

```
void AMyHUD::addWidget( Widget widget )
{
  // find the pos of the widget based on the grid.
  // draw the icons..
  FVector2D start( 200, 200 ), pad( 12, 12 );
  widget.size = FVector2D( 100, 100 );
  widget.pos = start;
  // compute the position here
  for( int c = 0; c < widgets.Num(); c++ )
  {
    // Move the position to the right a bit.
    widget.pos.X += widget.size.X + pad.X;
    // If there is no more room to the right then
    // jump to the next line
    if( widget.pos.X + widget.size.X > dims.X )
    {
      widget.pos.X = start.X;
      widget.pos.Y += widget.size.Y + pad.Y;
    }
  }
  widgets.Add( widget );
}

void AMyHUD::clearWidgets()
{
    widgets.Empty();
}
```

Make sure you add the following to the `.h` file as well:

```
void clearWidgets();
void addWidget(Widget widget);
```

8. We keep using the `Boolean` variable in `inventoryShowing` to tell us whether the inventory is currently displayed or not. When the inventory is shown, we also show the mouse so that the user knows what he's clicking on. Also, when the inventory is displayed, the free motion of the player is disabled. The easiest way to disable a player's free motion is by simply returning from the movement functions before actually moving. The following code is an example:

```
void AAvatar::Yaw( float amount )
{
  if( inventoryShowing )
  {
    return; // when my inventory is showing,
    // player can't move
  }
  AddControllerYawInput(200.f*amount * GetWorld()-
   >GetDeltaSeconds());
}
```

Exercise

Add `if( inventoryShowing ) { return; }` to each of the movement functions so when inventory is showing it will block all movement.

Detecting inventory item clicks

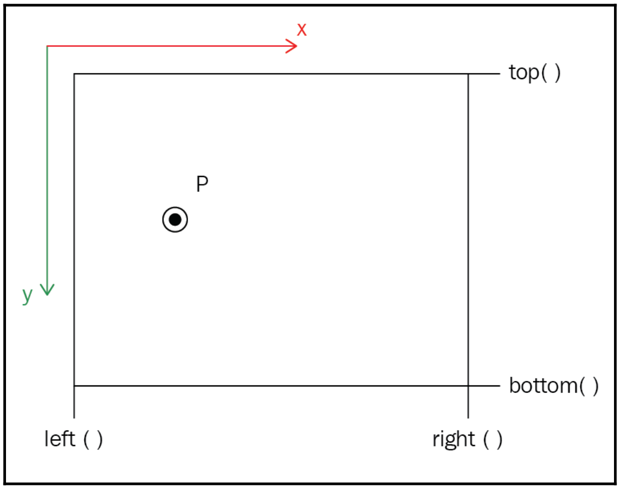
We can detect whether someone is clicking on one of our inventory items by doing a simple test to see if the point is inside the `rect` (rectangle) of an object. This test is done by checking the point of the click against the contents of the `rect` containing the area you want to test.

To check against the `rect`, add the following member function to `struct Widget`:

```
struct Widget
{
  // .. rest of struct same as before ..
  bool hit( FVector2D p )
  {
    // +---+ top (0)
    // |   |
    // +---+ bottom (2) (bottom > top)
    // L   R
    return p.X > left() && p.X < right() && p.Y > top() && p.Y <
```

```
        bottom();
    }
};
```

The test against the `rect` is as follows:



So, it is a hit if `p.X` is all of:

- Right of `left() (p.X > left())`
- Left of `right() (p.X < right())`
- Below `top() (p.Y > top())`
- Above `bottom() (p.Y < bottom())`

Remember that in UE4 (and UI rendering in general), the *y* axis is inverted. In other words, y goes down in UE4. This means that `top()` is less than `bottom()`, since the origin (the `(0, 0)` point) is at the top-left corner of the screen.

Dragging elements

We can drag elements easily:

1. The first step to enable dragging is to respond to the left mouse button click. First, we'll write the function to execute when the left mouse button is clicked. In the `Avatar.h` file, add the following prototype to the class declaration:

    ```
    void MouseClicked();
    ```

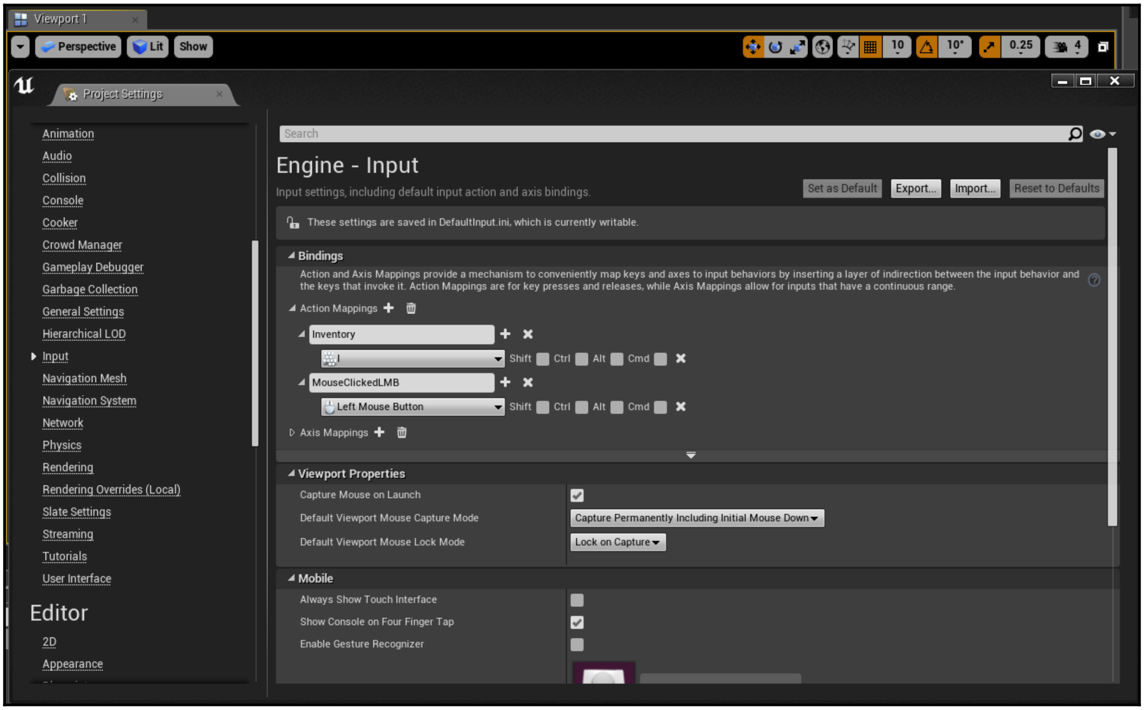
2. In the `Avatar.cpp` file, we can add a function to execute on a mouse click and pass the click request to the HUD, as follows:

```
void AAvatar::MouseClicked()
{
  APlayerController* PController = GetWorld()-
   >GetFirstPlayerController();
  AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
  hud->MouseClicked();
}
```

3. Then, in `AAvatar::SetupPlayerInputComponent`, we have to attach our responder:

```
PlayerInputComponent->BindAction( "MouseClickedLMB", IE_Pressed,
this, &AAvatar::MouseClicked );
```

The following screenshot shows how you set up the binding:



4. Add a member to the `AMyHUD` class, plus two new function definitions:

```
Widget* heldWidget;  // hold the last touched Widget in memory

void MouseClicked();
void MouseMoved();
```

5. Next, in `AMyHUD::MouseClicked()`, we start searching for the `Widget` hit:

```
void AMyHUD::MouseClicked()
{
    FVector2D mouse;
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();
    PController->GetMousePosition(mouse.X, mouse.Y);
    heldWidget = NULL; // clear handle on last held widget
                       // go and see if mouse xy click pos hits any
widgets
    for (int c = 0; c < widgets.Num(); c++)
    {
        if (widgets[c].hit(mouse))
        {
            heldWidget = &widgets[c];// save widget
            return;                  // stop checking
        }
    }
}
```

6. In the `AMyHUD::MouseClicked` function, we loop through all the widgets that are on the screen and check for a hit with the current mouse position. You can get the current mouse position from the controller at any time by simply looking up `PController->GetMousePosition()`.
7. Each widget is checked against the current mouse position, and the widget that got hit by the mouse click will be moved once the mouse is dragged. Once we have determined which widget got hit, we can stop checking, so we have a `return` value from the `MouseClicked()` function.
8. Hitting the widget is not enough, though. We need to drag the widget that got hit when the mouse moves. For this, we need to implement a `MouseMoved()` function in `AMyHUD`:

```
void AMyHUD::MouseMoved()
{
    static FVector2D lastMouse;
    FVector2D thisMouse, dMouse;
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();
```

```
            PController->GetMousePosition(thisMouse.X, thisMouse.Y);
            dMouse = thisMouse - lastMouse;
            // See if the left mouse has been held down for
            // more than 0 seconds. if it has been held down,
            // then the drag can commence.
            float time = PController->GetInputKeyTimeDown(
                EKeys::LeftMouseButton);
            if (time > 0.f && heldWidget)
            {
                // the mouse is being held down.
                // move the widget by displacement amt
                heldWidget->pos.X += dMouse.X;
                heldWidget->pos.Y += dMouse.Y; // y inverted
            }
            lastMouse = thisMouse;
        }
```

The drag function looks at the difference in the mouse position between the last frame and this frame, and moves the selected widget by that amount. A `static` variable (global with local scope) is used to remember the `lastMouse` position between the calls for the `MouseMoved()` function.

How can we link the mouse's motion to running the `MouseMoved()` function in `AMyHUD`? If you remember, we have already connected the mouse motion in the `Avatar` class. The two functions that we used were these:

- `AAvatar::Pitch()` (the y axis)
- `AAvatar::Yaw()` (the x axis)

Extending these functions will enable you to pass mouse inputs to the HUD. I will show you the `Yaw` function now, and you can extrapolate how `Pitch` will work from there:

```
void AAvatar::Yaw( float amount )
{
  //x axis
  if( inventoryShowing )
  {
    // When the inventory is showing,
    // pass the input to the HUD
    APlayerController* PController = GetWorld()-
     >GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
    hud->MouseMoved();
    return;
  }
  else
```

```
    {
      AddControllerYawInput(200.f*amount * GetWorld()-
        >GetDeltaSeconds());
    }
  }
```

The `AAvatar::Yaw()` function first checks whether the inventory is showing or not. If it is showing, inputs are routed straight to the HUD, without affecting `Avatar`. If the HUD is not showing, inputs just go to `Avatar`.

Make sure you added `#include "MyHUD.h"` to the top of the file for this to work.

Exercises

1. Complete the `AAvatar::Pitch()` function (y axis) to route inputs to the HUD instead of to `Avatar`.
2. Take the NPC characters from `Chapter 8`, *Actors and Pawns*, and give the player an item (such as `GoldenEgg`) when he goes near them.

Putting things together

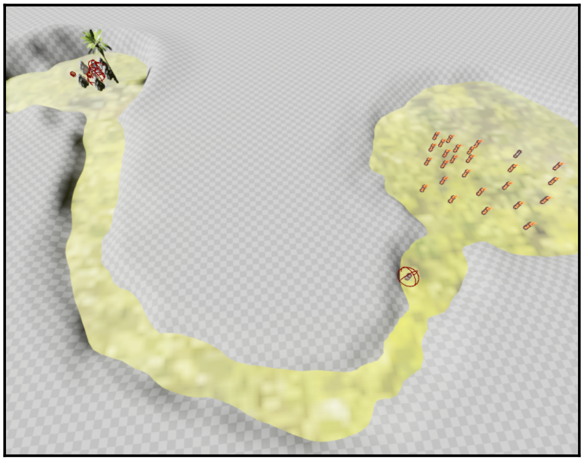
Now that you have all this code, you'll want to put this together and see it working. Use the Meshes you copied over to create new blueprints by right-clicking the `PickupItem` class in the Class Viewer and select **Create Blueprint Class** as we did previously. Set the values (including the Mesh) and then drag objects into the game. When you walk into them, you will get a message that it was picked up. At that point, you can hit *I* to view your inventory.

Summary

In this chapter, we covered how to set up multiple pickup items for the player to see displayed in the level and also pick up. We also displayed them on screen and added functionality to drag the widgets. In `Chapter 11`, *Monsters*, we will introduce Monsters and how to make them follow and attack the player.

11
Monsters

In this chapter, we will be adding opponents for the player. We'll be creating a new landscape to roam around in, with monsters that will start walking toward the player when they are close enough to detect them. They will also attack once they get within range of the player, giving you some basic gameplay.



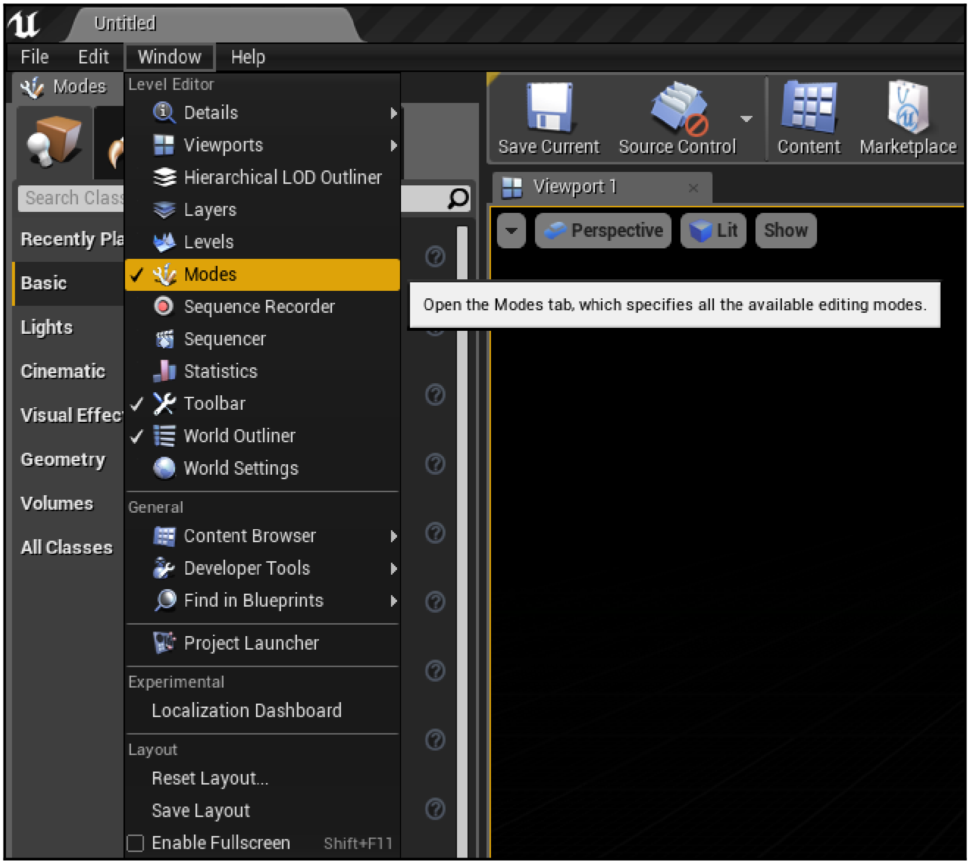
Let's take a look at the topics covered in this chapter:

- Landscape
- Creating Monsters
- Monster attacks on the player

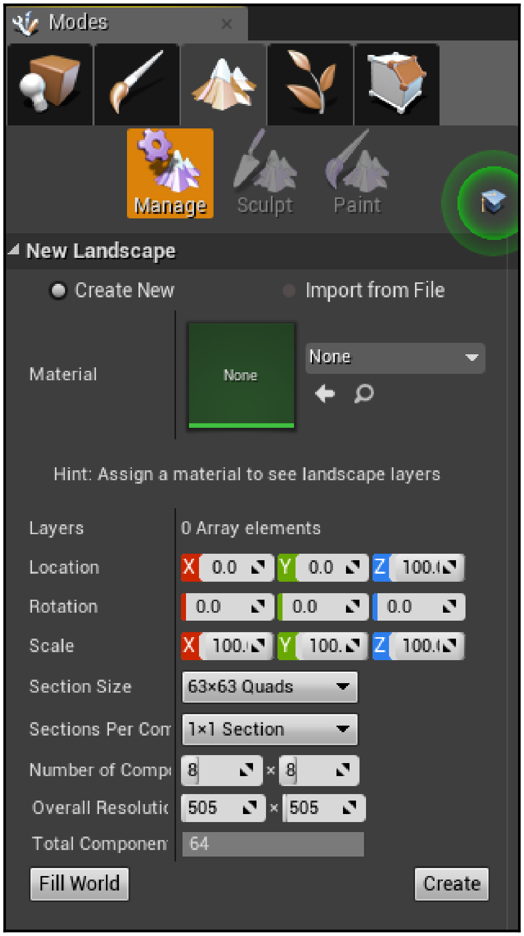
Landscape

We haven't covered how to sculpt the landscape in this book yet, so we'll do that here. First, you must have a landscape to work with. To do that, follow these steps:

1. Start a new file by navigating to **File** | **New Level...**. You can choose an empty level or a level with a sky. I chose the one without the sky in this example.
2. To create a landscape, we have to work from the **Modes** panel. Make sure that the **Modes** panel is displayed by navigating to **Window** | **Modes**:



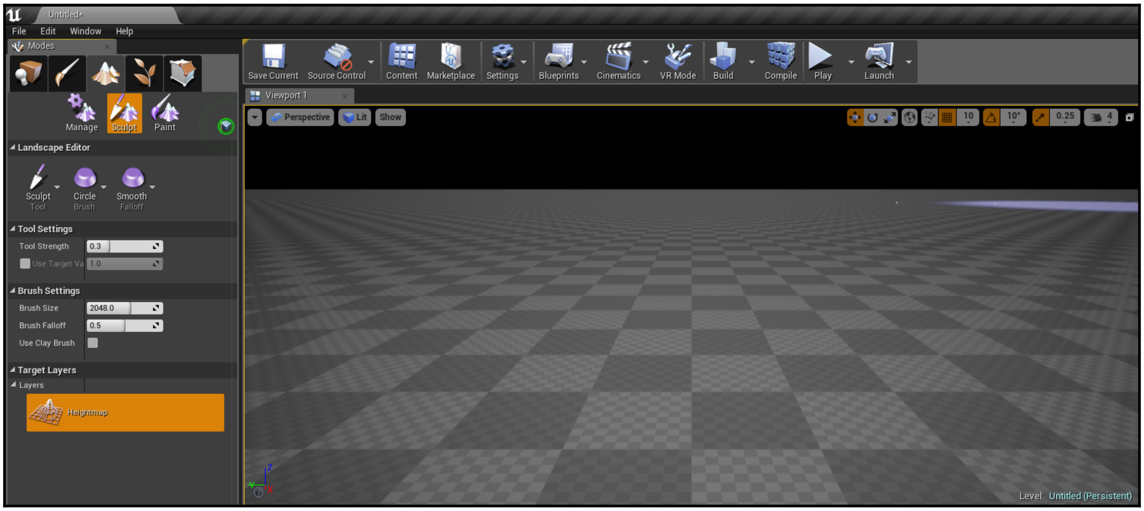
3. A landscape can be created in three steps, which are shown in the following screenshot:



The three steps are as follows:

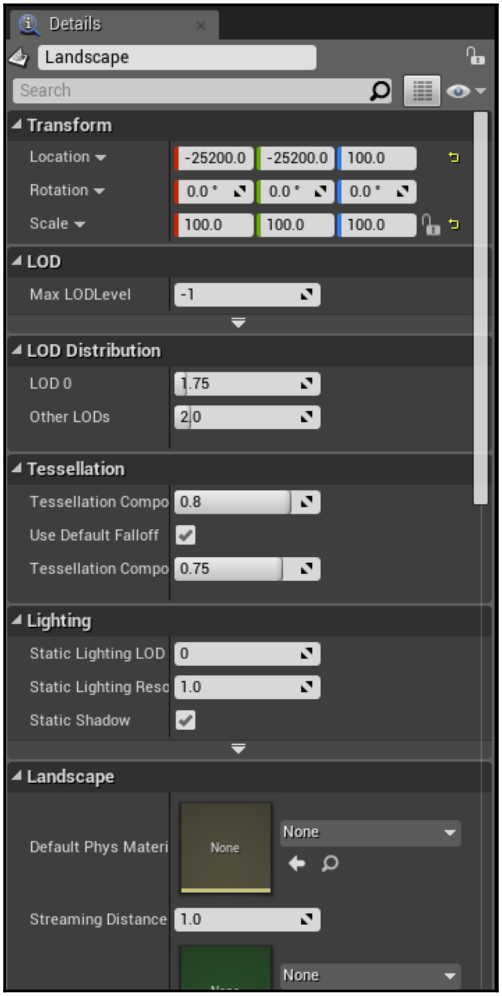
1. Click on the landscape icon (the picture of the mountains) in the **Modes** panel
2. Click on the **Manage** button
3. Click on the **Create** button in the lower right-hand corner of the screen

4. You should now have a landscape to work with. It will appear as a gray, tiled area in the main window:



The first thing you will want to do with your landscape scene is add some color to it. What's a landscape without colors?

5. Click anywhere on your gray, tiled landscape object. In the **Details** panel on the right, you will see that it is populated with information, as shown in the following screenshot:

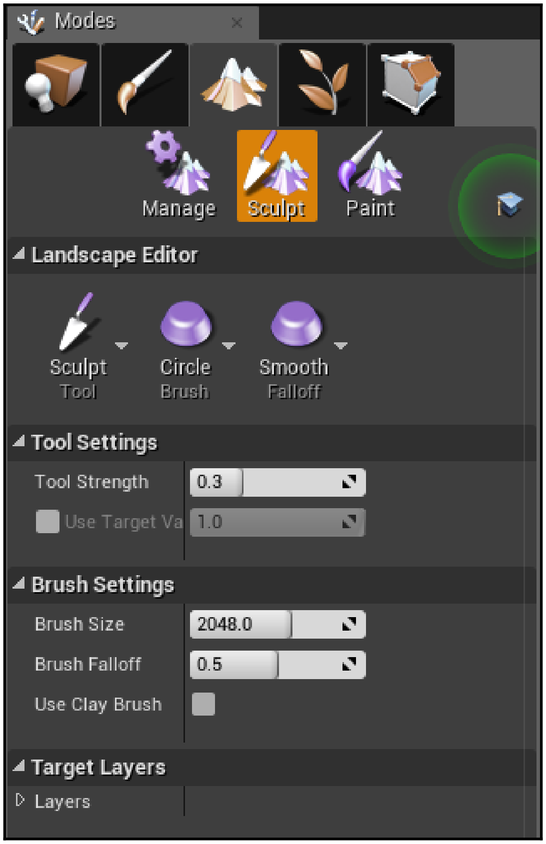


6. Scroll down until you see the **Landscape Material** property. You can select the **M_Ground_Grass** material for a realistic-looking ground.
7. Add a light to the scene. You should probably use a directional light so that all of the ground has some light on it. We went over how to do this in `Chapter 8`, *Actors and Pawns*.

Sculpting the landscape

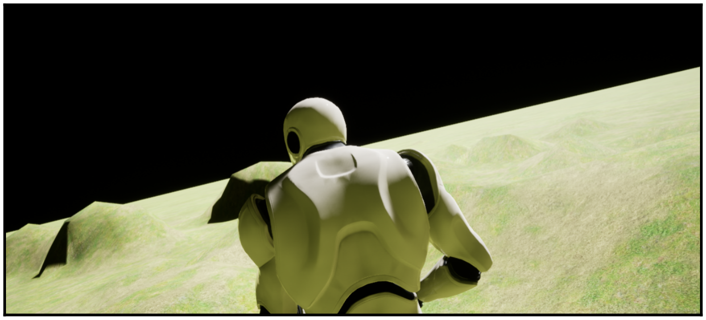
A flat landscape can be boring. We should at least add some curves and hills to the place. To do so, perform the following steps:

1. Click on the **Sculpt** button in the **Modes** panel:

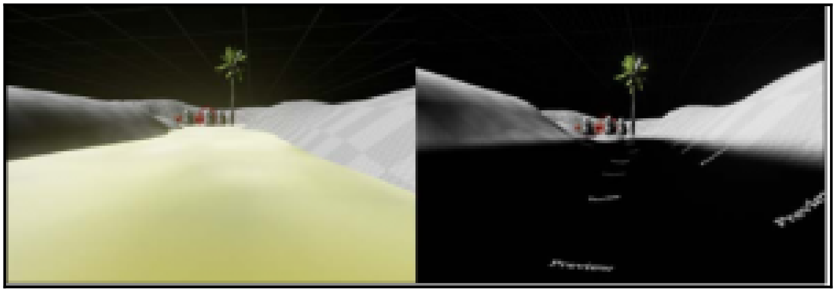


The strength and size of your brush are determined by the **Brush Size** and **Tool Strength** parameters in the **Modes** window.

2. Click on your landscape and drag the mouse to change the height of the turf.
3. Once you're happy with what you've got, click on the **Play** button to try it out. The resultant output can be seen in the following screenshot:



4. Play around with your landscape and create a scene. What I did was lower the landscape around a flat ground plane so that the player has a well-defined flat area to walk on, as shown in the following screenshot:



Feel free to do whatever you like with your landscape. You can use what I'm doing here as inspiration, if you like.

I recommend that you import assets from **ContentExamples** or from **StrategyGame** so that you can use them inside your game. To do this, refer to the *Importing assets* section in `Chapter 10`, *Inventory System and Pickup Items*. When you're done importing assets, we can proceed to bringing monsters into our world.

Creating Monsters

We'll start programming monsters in the same way we programmed NPCs and `PickupItem`. We will write a base class (by deriving from character) to represent the `Monster` class, then derive a bunch of blueprints for each monster type. Each monster will have a couple of properties in common that determine its behavior. The following are the common properties:

- It will have a `float` variable for speed.
- It will have a `float` variable for the `HitPoints` value (I usually use floats for HP, so we can easily model HP leeching effects such as walking through a pool of lava).
- It will have an `int32` variable for the experience gained in defeating the monster.
- It will have a `UClass` function for the loot dropped by the monster.
- It will have a `float` variable for the `BaseAttackDamage` done by each attack.
- It will have a `float` variable for `AttackTimeout`, which is the amount of time for which the monster rests between attacking
- It will have two `USphereComponents` objects: one of them is `SightSphere`—how far the monster can see. The other is `AttackRangeSphere`, which is how far its attack reaches. The `AttackRangeSphere` object is always smaller than `SightSphere`.

Follow these steps:

1. Derive from the `Character` class to create your class for `Monster`. You can do this in UE4 by going to **File | New C++ Class...** and then selecting the **Character** option from the menu for your base class.
2. Fill out the `Monster` class with the base properties.
3. Make sure that you declare `UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)` so that the properties of the monsters can be changed in the blueprints. This is what you should have in `Monster.h`:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "Components/SphereComponent.h"
#include "Monster.generated.h"

UCLASS()
```

```
class GOLDENEGG_API AMonster : public ACharacter
{
    GENERATED_BODY()
public:
    AMonster(const FObjectInitializer& ObjectInitializer);

        // How fast he is
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
            MonsterProperties)
        float Speed;

    // The hitpoints the monster has
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        MonsterProperties)
        float HitPoints;

    // Experience gained for defeating
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        MonsterProperties)
        int32 Experience;

    // Blueprint of the type of item dropped by the monster
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        MonsterProperties)
        UClass* BPLoot;

    // The amount of damage attacks do
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        MonsterProperties)
        float BaseAttackDamage;

    // Amount of time the monster needs to rest in seconds
    // between attacking
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        MonsterProperties)
        float AttackTimeout;

    // Time since monster's last strike, readable in blueprints
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
        MonsterProperties)
        float TimeSinceLastStrike;

    // Range for his sight
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category
=
        Collision)
        USph.ereComponent* SightSphere;
```

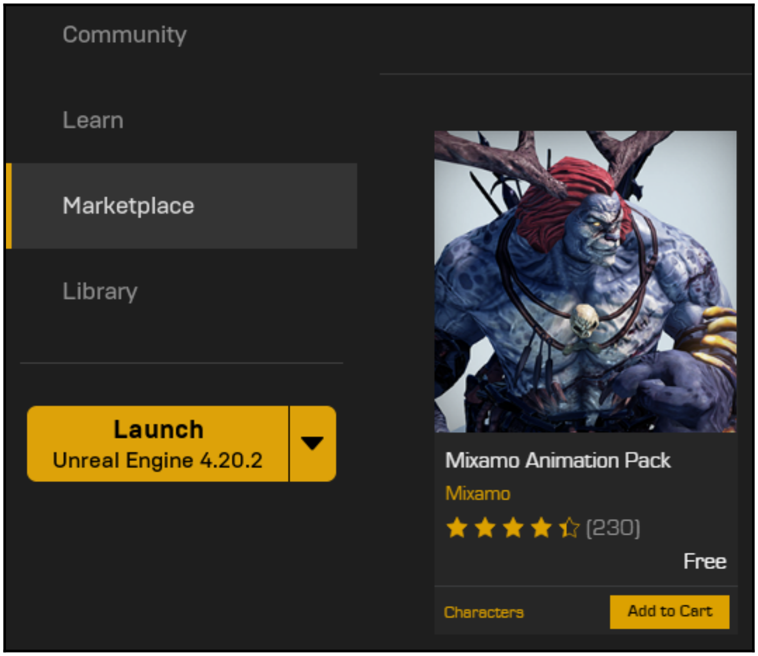
```
        // Range for his attack. Visualizes as a sphere in editor,
        UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category
    =
            Collision)
            USphereComponent* AttackRangeSphere;
    };
```

4. You will need some bare minimum code in your `Monster` constructor to get the monster's properties initialized. Use the following code in the `Monster.cpp` file (this should replace the default constructor):

```
AMonster::AMonster(const FObjectInitializer& ObjectInitializer)
 : Super(ObjectInitializer)
{
 Speed = 20;
 HitPoints = 20;
 Experience = 0;
 BPLoot = NULL;
 BaseAttackDamage = 1;
 AttackTimeout = 1.5f;
 TimeSinceLastStrike = 0;

 SightSphere =
ObjectInitializer.CreateDefaultSubobject<USphereComponent>
 (this, TEXT("SightSphere"));
 SightSphere->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepWorldTransform);

 AttackRangeSphere = ObjectInitializer.CreateDefaultSubobject
 <USphereComponent>(this, TEXT("AttackRangeSphere"));
 AttackRangeSphere->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepWorldTransform);
}
```

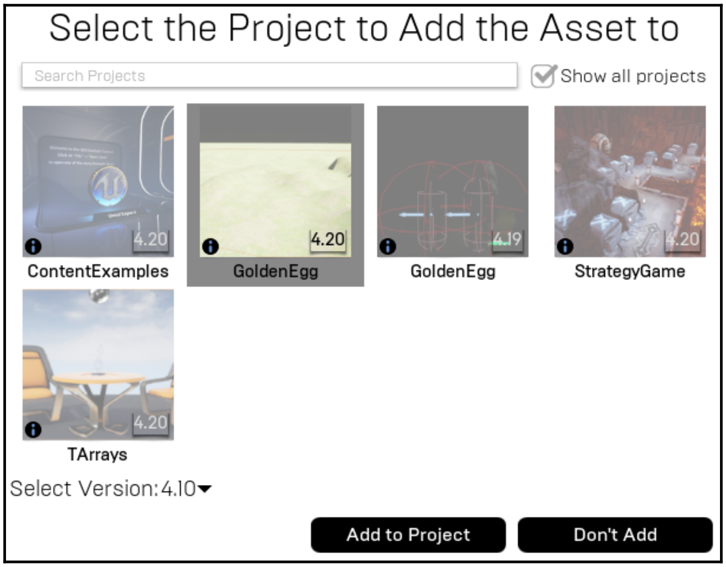
5. Compile and run the code.
6. Open Unreal Editor and derive a blueprint based on your `Monster` class (call it `BP_Monster`).
7. Now, we can start configuring our monster's `Monster` properties. For the skeletal mesh, we won't use the same model for the monster because we need the monster to be able to do melee attacks, and the same model does not come with a melee attack. However, some of the models in the **Mixamo Animation Pack** file have melee attack animations.

8. So, download the **Mixamo Animation Pack** file from the UE4 marketplace (free):

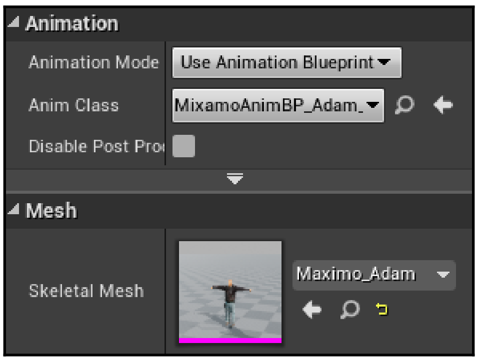


Inside the pack are some pretty gross models that I'd avoid, but others are quite good.

9. You should add the **Mixamo Animation Pack** file to your project. It hasn't been updated for a while, but you can add it by checking **Show all projects** and selecting version 4.10 from the drop-down list, as shown in the following screenshot:



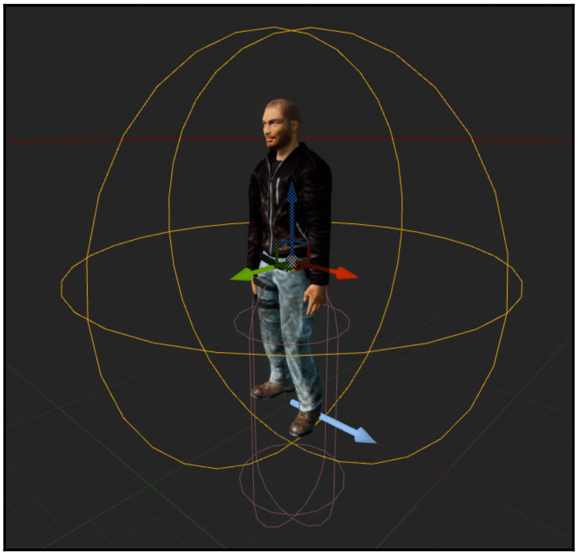
10. Edit the `BP_Monster` blueprint's class properties and select **Mixamo_Adam** (it is actually typed as **Maximo_Adam** in the current issue of the package) as the skeletal mesh. Make sure that you line it up with the capsule component. Also, select **MixamoAnimBP_Adam** as the animation blueprint:



We'll modify the animation blueprint to correctly incorporate the melee attack animation later.

> **TIP**
> While you're editing your `BP_Monster` blueprint, change the sizes of the `SightSphere` and `AttackRangeSphere` objects to values that make sense to you. I made my monster's `AttackRangeSphere` object just big enough to be about an arm's reach (60 units) and his `SightSphere` object to be 25 times bigger than that (about 1,500 units).

Remember that the monster will start moving toward the player once he enters the monster's `SightSphere`, and the monster will start attacking the player once he is inside the monster's `AttackRangeSphere` object:



Place a few of your `BP_Monster` instances inside your game; compile and run. Without any code to drive the `Monster` character to move, your monsters should just stand there idly.

Basic monster intelligence

In our game, we'll add only basic intelligence to the `Monster` characters. The monsters will know how to do two basic things:

- Track the player and follow him
- Attack the player

The monster won't do anything else. You can have the monster taunt the player when the player is first seen as well, but we'll leave that as an exercise for you.

Moving the monster – steering behavior

Monsters in very basic games don't usually have complex motion behaviors. Usually, they just walk toward the target and attack it. We'll program that type of monster in this game, but you can get more interesting play with monsters that position themselves advantageously on the terrain to perform ranged attacks and so on. We're not going to program that here, but it's something to think about.

In order to get the `Monster` character to move toward the player, we need to dynamically update the direction of the `Monster` character moving in each frame. To update the direction that the monster is facing, we write code in the `Monster::Tick()` method.

The `Tick` function runs in every frame of the game. The signature of the `Tick` function is as follows:

```
virtual void Tick(float DeltaSeconds) override;
```

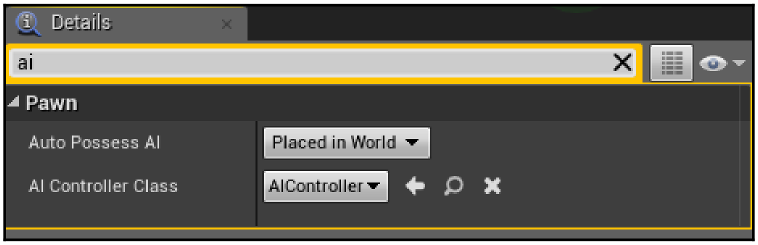
You need to add this function's prototype to your `AMonster` class in your `Monster.h` file. If we override `Tick`, we can place our own custom behavior that the `Monster` character should do in each frame. Here's some basic code that will move the monster toward the player during each frame:

```
void AMonster::Tick(float DeltaSeconds) {
    Super::Tick(DeltaSeconds);
    //basic intel : move the monster towards the player
    AAvatar *avatar = Cast<AAvatar>(
            UGameplayStatics::GetPlayerPawn(GetWorld(), 0));
    if (!avatar) return;
    FVector toPlayer = avatar->GetActorLocation() - GetActorLocation();
    toPlayer.Normalize(); // reduce to unit vector
                        // Actually move the monster towards the player a
bit
    AddMovementInput(toPlayer, Speed*DeltaSeconds); // At least face the
target
    // Gets you the rotator to turn something // that looks in the
`toPlayer`direction
    FRotator toPlayerRotation = toPlayer.Rotation();
    toPlayerRotation.Pitch = 0; // 0 off the pitch
    RootComponent->SetWorldRotation(toPlayerRotation);
}
```

You also have to add the following includes at the top of the file:

```
#include "Avatar.h"

#include "Kismet/GameplayStatics.h"
```

For `AddMovementInput` to work, you must have a controller selected under the **AIController Class** panel in your blueprint, as shown in the following screenshot:



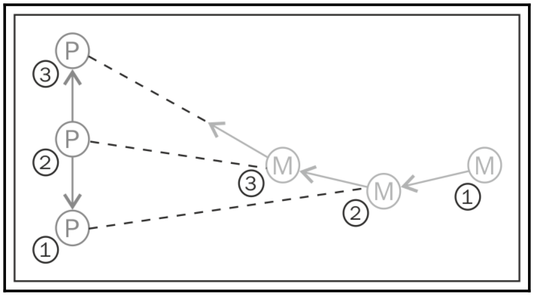
If you have selected `None`, calls to `AddMovementInput` won't have any effect. To prevent this, select either the `AIController` class or the `PlayerController` class as your **AIController Class**. Make sure that you check this for every monster you've placed on the map.

The preceding code is very simple. It comprises the most basic form of enemy intelligence—simply move toward the player by an incrementally small amount in each frame:

TIP

If your monsters are facing away from the player, try changing the rotation of the mesh -90 degrees in the Z direction.

The result, after a series of frames, will be that the monster tracks and follows the player around the level. To understand how this works, you must remember that the `Tick` function is called on average about 60 times per second. What this means is that, in each frame, the monster moves a tiny bit closer to the player. Since the monster moves in very small steps, its action looks smooth and continuous (while in reality, it is making small jumps and leaps in each frame):



Discrete nature of tracking—a monster's motion over three superimposed frames

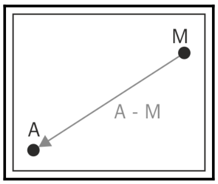
The reason why the monster moves about 60 times a second is because of a hardware constraint. The refresh rate of a typical monitor is 60 Hz, so it acts as a practical limiter on how many updates per second are useful. Updating at a frame rate faster than the refresh rate is possible, but it is not necessarily useful for games since you will only see a new picture once every 1/60 of a second on most hardware. Some advanced physics modeling simulations do almost 1,000 updates a second, but arguably, you don't need that kind of resolution for a game and you should reserve the extra CPU time for something that the player will enjoy instead, such as better AI algorithms. Some newer hardware boasts of a refresh rate of up to 120 Hz (look up gaming monitors, but don't tell your parents I asked you to blow all your money on one).

The discrete nature of monster motion

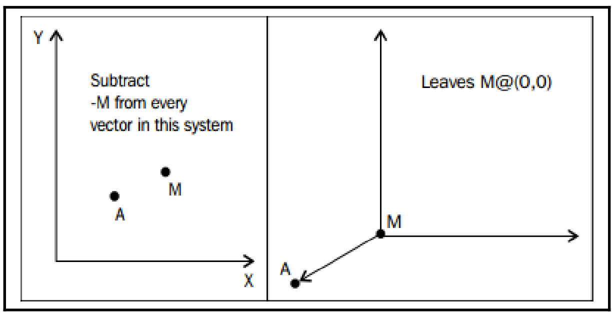
Computer games are discrete in nature. In the preceding screenshot of superimposed sequences of frames, the player is seen as moving straight up the screen, in tiny steps. The motion of the monster is also in small steps. In each frame, the monster takes one small discrete step toward the player. The monster is following an apparently curved path as it moves directly toward where the player is in each frame.

To move the monster toward the player, follow these steps:

1. We have to get the player's position. Since the player is accessible in a global function, `UGameplayStatics::GetPlayerPawn`, we simply retrieve our pointer to the player using this function.
2. We find the vector pointing from the `Monster` function (`GetActorLocation()`) that points to the player (`avatar->GetActorLocation()`).
3. We need to find the vector that points from the monster to the avatar. To do this, you have to subtract the location of the monster from the location of the avatar, as shown in the following screenshot:



It's a simple math rule to remember, but often easy to get wrong. To get the right vector, always subtract the source (the starting point) vector from the target (the terminal point) vector. In our system, we have to subtract the `Monster` vector from the `Avatar` vector. This works because subtracting the `Monster` vector from the system moves the `Monster` vector to the origin, and the `Avatar` vector will be to the lower left-hand side of the `Monster` vector:



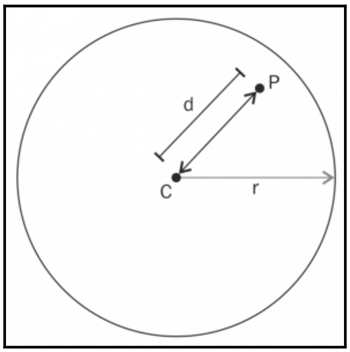
Be sure to try out your code. At this point, the monsters will be running toward your player and crowding around him. With the preceding code that is outlined, they won't attack; they'll just follow him around, as shown in the following screenshot:

Monster SightSphere

Right now, the monsters are not paying attention to the `SightSphere` component. That is, wherever the player is in the world, the monsters will move toward him in the current setup. We want to change that now.

To do so, all we have to do is have `Monster` respect the `SightSphere` restriction. If the player is inside the monster's `SightSphere` object, the monster will give chase. Otherwise, the monsters will be oblivious to the player's location and not chase the player.

Checking to see if an object is inside a sphere is simple. In the following screenshot, the point **p** is inside the sphere if the distance **d** between **p** and the center **c** is less than the sphere radius, **r**:



P is inside the sphere when d is less than r

So, in our code, the preceding screenshot translates to the following:

```
void AMonster::Tick(float DeltaSeconds)
{
  Super::Tick( DeltaSeconds );
  AAvatar *avatar = Cast<AAvatar>(
   UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
  if( !avatar ) return;
    FVector toPlayer = avatar->GetActorLocation() -
     GetActorLocation();
  float distanceToPlayer = toPlayer.Size();
  // If the player is not in the SightSphere of the monster,
  // go back
  if( distanceToPlayer > SightSphere->GetScaledSphereRadius() )
  {
    // If the player is out of sight,
    // then the enemy cannot chase
    return;
  }

  toPlayer /= distanceToPlayer;  // normalizes the vector
  // Actually move the monster towards the player a bit
  AddMovementInput(toPlayer, Speed*DeltaSeconds);
  // (rest of function same as before (rotation))
}
```

The preceding code adds additional intelligence to the `Monster` character. The `Monster` character can now stop chasing the player if the player is outside the monster's `SightSphere` object. This is how the result will look:

A good thing to do here will be to wrap up the distance comparison into a simple inline function. We can provide these two inline member functions in the `Monster` header, as follows:

```
inline bool isInSightRange( float d )
{ return d < SightSphere->GetScaledSphereRadius(); }
inline bool isInAttackRange( float d )
{ return d < AttackRangeSphere->GetScaledSphereRadius(); }
```

These functions return the value `true` when the passed parameter, `d`, is inside the spheres in question.

> TIP: An inline function means that the function is more like a macro than a function. Macros are copied and pasted to the calling location, while functions are jumped to by C++ and executed at their location. Inline functions are good because they give good performance while keeping the code easy to read. They are reusable.

Monster attacks on the player

There are a few different types of attacks that monsters can do. Depending on the type of the `Monster` character, a monster's attack might be melee (close range) or ranged (projectile weapon).

The `Monster` character will attack the player whenever the player is in its `AttackRangeSphere` object. If the player is out of the range of the monster's `AttackRangeSphere` object but the player is in the `SightSphere` object of the monster, then the monster will move closer to the player until the player is in the monster's `AttackRangeSphere` object.

Melee attacks

The dictionary definition of *melee* is a confused mass of people. A melee attack is one that is done at a close range. Picture a bunch of *zerglings* battling it out with a bunch of *ultralisks* (if you're a StarCraft player, you'll know that both zerglings and ultralisks are melee units). Melee attacks are basically close range, hand-to-hand combat. To do a melee attack, you need a melee attack animation that turns on when the monster begins its melee attack. To do this, you need to edit the animation blueprint in UE4's animation editor.

> **TIP** Zak Parrish's series is an excellent place to get started with programming animations in blueprints:
> `https://www.youtube.com/watch?v=AqYmC2wn7Cg&list=PL6VDVOqa_mdNW6JEu9UAS_s40OCD_u6yp&index=8`.

For now, we'll just program the melee attack and then worry about modifying the animation in blueprints later.

Defining a melee weapon

There are going to be three parts to defining our melee weapon. They are as follows:

- The C++ code that represents it
- The model
- A UE4 blueprint that connects the code and model together

Coding for a melee weapon in C++

We'll define a new class, `AMeleeWeapon` (derived from `AActor`), to represent hand-held combat weapons (as you might have figured out by now, the A is added automatically to the name you use). I will attach a couple of blueprint-editable properties to the `AMeleeWeapon` class, and the `AMeleeWeapon` class will look as shown in the following code:

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Components/BoxComponent.h"
#include "MeleeWeapon.generated.h"

class AMonster;

UCLASS()
class GOLDENEGG_API AMeleeWeapon : public AActor
{
    GENERATED_BODY()
public:
    AMeleeWeapon(const FObjectInitializer& ObjectInitializer);

    // The amount of damage attacks by this weapon do
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
        MeleeWeapon)
        float AttackDamage;

    // A list of things the melee weapon already hit this swing
```

```
    // Ensures each thing sword passes thru only gets hit once
    TArray<AActor*> ThingsHit;

    // prevents damage from occurring in frames where
    // the sword is not swinging
    bool Swinging;

    // "Stop hitting yourself" - used to check if the
    // actor holding the weapon is hitting himself
    AMonster *WeaponHolder;

    // bounding box that determines when melee weapon hit
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
        MeleeWeapon)
        UBoxComponent* ProxBox;

    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
        MeleeWeapon)
        UStaticMeshComponent* Mesh;

    UFUNCTION(BlueprintNativeEvent, Category = Collision)
        void Prox(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp,
            int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);

    // You shouldn't need this unless you get a compiler error that it
can't find this function.
    virtual int Prox_Implementation(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp,
        int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);

    void Swing();
    void Rest();
};
```

Notice how I used a bounding box for `ProxBox` and not a bounding sphere. This is because swords and axes will be better approximated by boxes rather than spheres. There are two member functions, `Rest()` and `Swing()`, which let `MeleeWeapon` know what state the actor is in (resting or swinging). There's also a `TArray<AActor*> ThingsHit` property inside this class that keeps track of the actors hit by this melee weapon on each swing. We are programming it so that the weapon can only hit each thing once per swing.

The `AMeleeWeapon.cpp` file will contain just a basic constructor and some simple code to send damages to `OtherActor` when our sword hits it. We'll also implement the `Rest()` and `Swing()` functions to clear the list of things hit. The `MeleeWeapon.cpp` file has the following code:

```
#include "MeleeWeapon.h"
#include "Monster.h"


AMeleeWeapon::AMeleeWeapon(const FObjectInitializer& ObjectInitializer)
    : Super(ObjectInitializer)
{
    AttackDamage = 1;
    Swinging = false;
    WeaponHolder = NULL;

    Mesh =
ObjectInitializer.CreateDefaultSubobject<UStaticMeshComponent>(this,
        TEXT("Mesh"));
    RootComponent = Mesh;

    ProxBox = ObjectInitializer.CreateDefaultSubobject<UBoxComponent>(this,
        TEXT("ProxBox"));
    ProxBox->OnComponentBeginOverlap.AddDynamic(this,
            &AMeleeWeapon::Prox);
    ProxBox->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepWorldTransform);
}

int AMeleeWeapon::Prox_Implementation(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // don't hit non root components
    if (OtherComp != OtherActor->GetRootComponent())
    {
        return -1;
    }

    // avoid hitting things while sword isn't swinging,
    // avoid hitting yourself, and
    // avoid hitting the same OtherActor twice
    if (Swinging && OtherActor != (AActor *) WeaponHolder &&
        !ThingsHit.Contains(OtherActor))
    {
        OtherActor->TakeDamage(AttackDamage +
WeaponHolder->BaseAttackDamage, FDamageEvent(), NULL, this);
```

```
        ThingsHit.Add(OtherActor);
    }

    return 0;
}

void AMeleeWeapon::Swing()
{
    ThingsHit.Empty();  // empty the list
    Swinging = true;
}

void AMeleeWeapon::Rest()
{
    ThingsHit.Empty();
    Swinging = false;
}
```

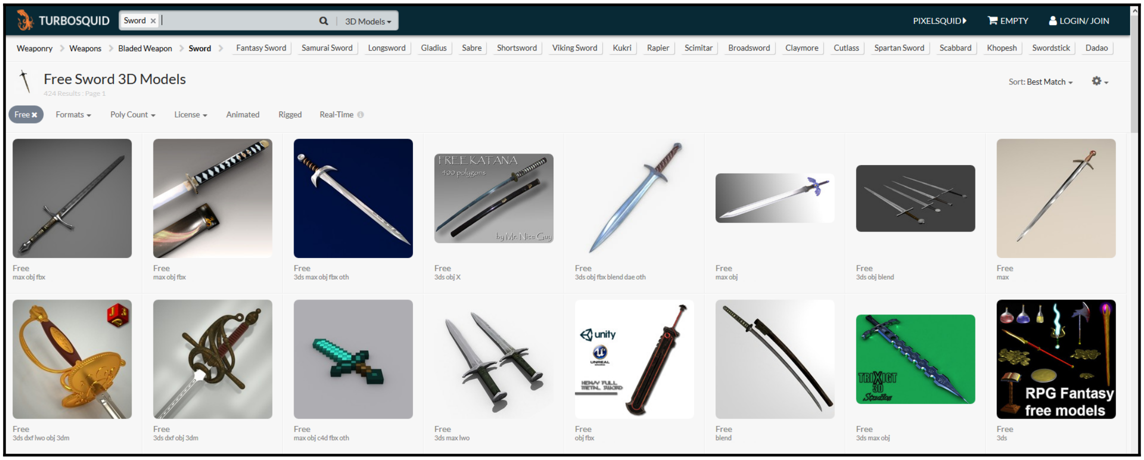
Downloading a sword

To complete this exercise, we need a sword to put into the model's hand. I added a sword to the project called *Kilic* from `http://tf3dm.com/3d-model/sword-95782.html` by Kaan Gülhan. The following is a list of other places where you will get free models:

- `http://www.turbosquid.com/`
- `http://tf3dm.com/`
- `http://archive3d.net/`
- `http://www.3dtotal.com/`

Secret tip

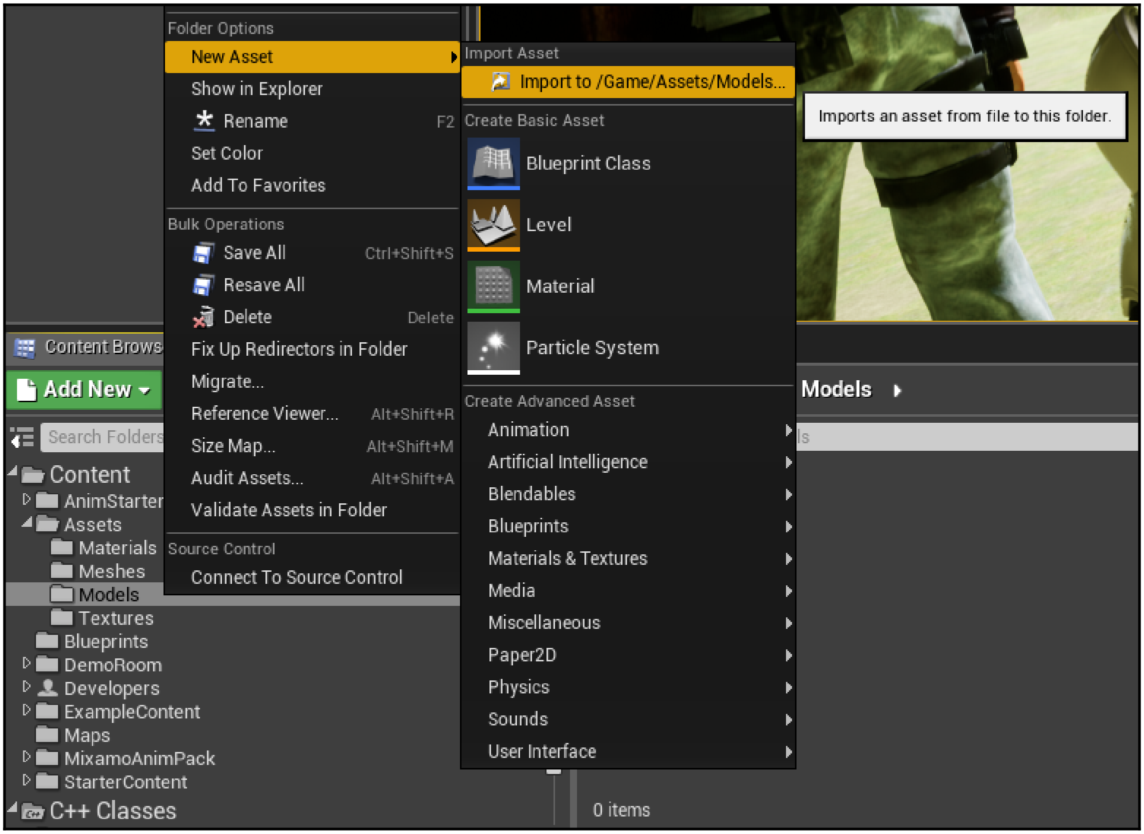
It might appear at first on `TurboSquid.com` that there are no free models. In fact, the secret is that you have to select free under **Price**:



I had to edit the kilic sword mesh slightly to fix the initial sizing and rotation. You can import any mesh in the **Filmbox** (**FBX**) format into your game. The kilic sword model is in the sample code package for this chapter. To import your sword into the UE4 editor, perform the following steps:

1. Right-click on any folder you want to add the model to
2. Navigate to **New Asset** | **Import to (path)...**
3. From the file explorer that pops up, select the new asset you want to import.
4. If a **Models** folder doesn't exist, you can create one by simply right-clicking on the tree view at the left and selecting **New Folder** in the pane on the left-hand side of the **Content Browser** tab

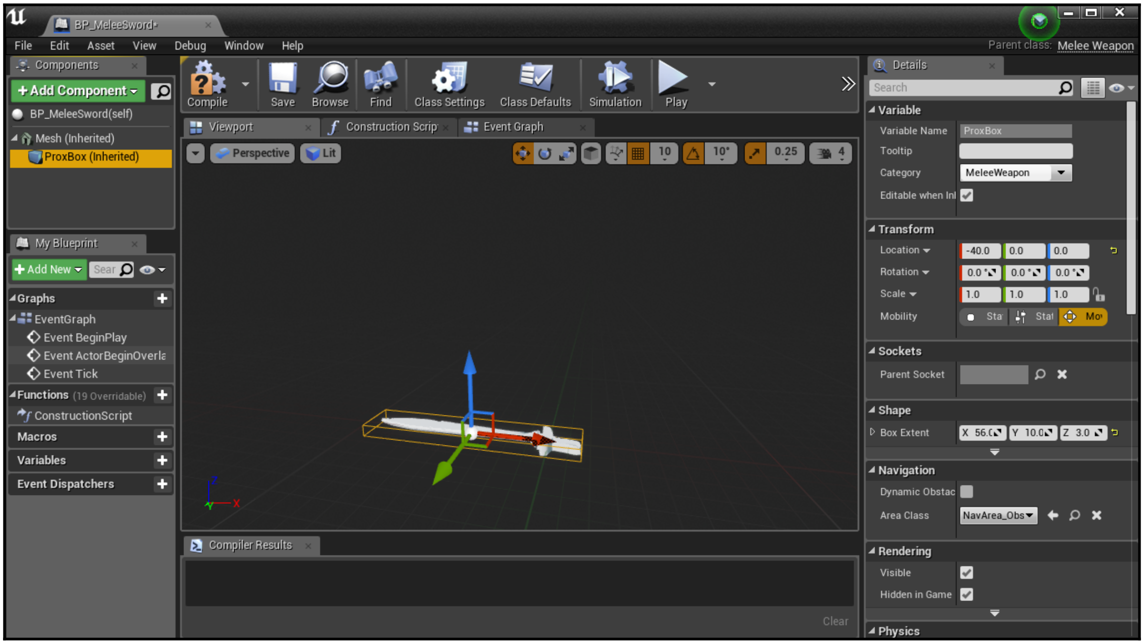
I selected the `kilic.fbx` asset from my desktop:



Creating a blueprint for your melee weapon

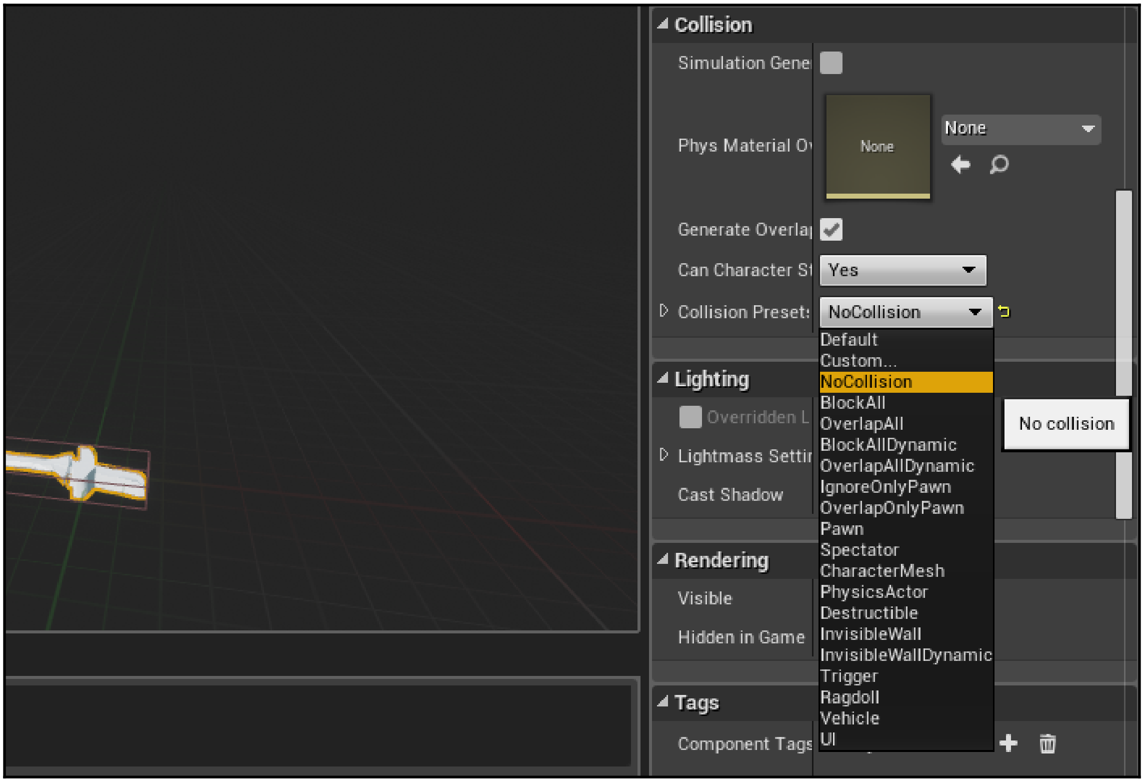
The steps to create a blueprint for your melee weapon are as follows:

1. Inside the UE4 editor, create a blueprint based off of `AMeleeWeapon` called `BP_MeleeSword`.
2. Configure `BP_MeleeSword` to use the kilic blade model (or any blade model you choose), as shown in the following screenshot:



3. The ProxBox class will determine whether something was hit by the weapon, so we'll modify the ProxBox class so that it just encloses the blade of the sword, as shown in the following screenshot:

4. Under the **Collision Presets** panel, it is important to select the **NoCollision** option for the mesh (not **BlockAll**). This is illustrated in the following screenshot:



5. If you select **BlockAll**, then the game engine will automatically resolve all the interpenetration between the sword and the characters by pushing away things that the sword touches whenever it is swung. The result is that your characters will appear to go flying whenever a sword is swung.

Sockets

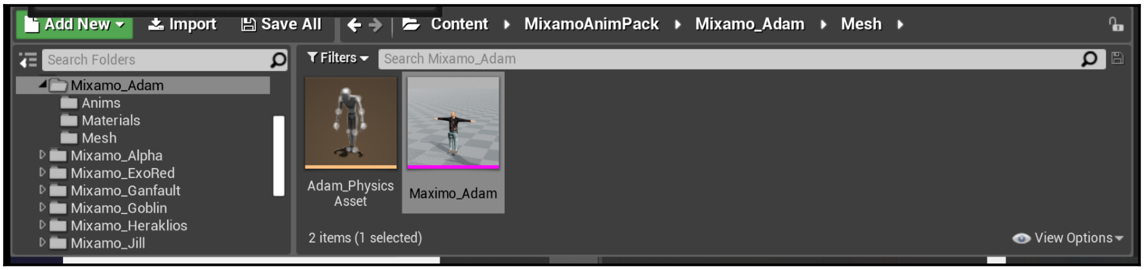
A socket in UE4 is a receptacle on one skeletal mesh for another `Actor`. You can place a socket anywhere on a skeletal mesh body. After you have correctly placed the socket, you can attach another `Actor` to this socket in UE4 code.

For example, if we want to put a sword in our monster's hand, we'd just have to create a socket in our monster's hand. We can attach a helmet to the player by creating a socket on his head.

Creating a skeletal mesh socket in the monster's hand

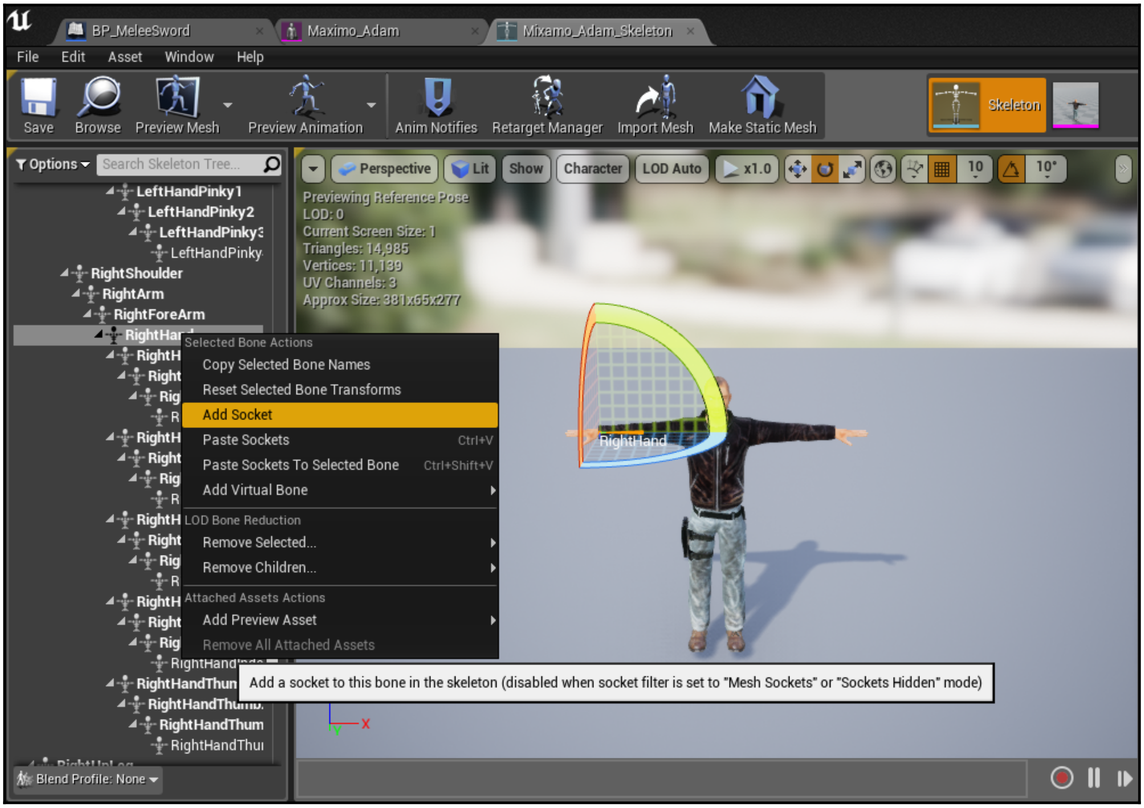
To attach a socket to the monster's hand, we have to edit the skeletal mesh that the monster is using. Since we used the **Mixamo_Adam** skeletal mesh for the monster, we have to open and edit this skeletal mesh. To do so, perform the following steps:

1. Double-click on the **Mixamo_Adam** skeletal mesh in the **Content Browser** tab (this will appear as the T-pose) to open the skeletal mesh editor.
2. If you don't see **Mixamo Adam** in your **Content Browser** tab, make sure that you have imported the **Mixamo Animation Pack** file into your project from the Unreal Launcher app:



3. Click on **Skeleton** at the top-right corner of the screen.
4. Scroll down the tree of bones in the left-hand side panel until you find the **RightHand** bone.

5. We'll attach a socket to this bone. Right-click on the **RightHand** bone and select **Add Socket**, as shown in the following screenshot:



6. You can leave the default name (**RightHandSocket**) or rename the socket if you like, as shown in the following screenshot:

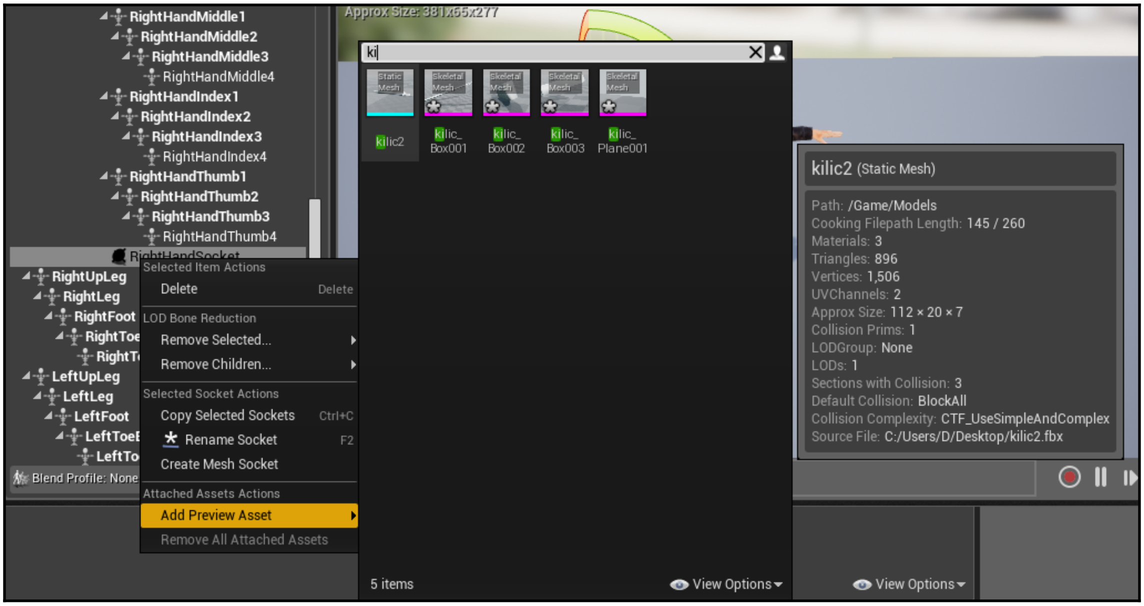
Next, we need to add a sword to the actor's hand.

Attaching the sword to the model

The steps to attach the sword are as follows:

1. With the Adam skeletal mesh open, find the **RightHandSocket** option in the tree view. Since Adam swings with his right hand, you should attach the sword to his right hand.

2. Right-click on the **RightHandSocket** option, select **Add Preview Asset**, and find the skeletal mesh for the sword in the window that appears:



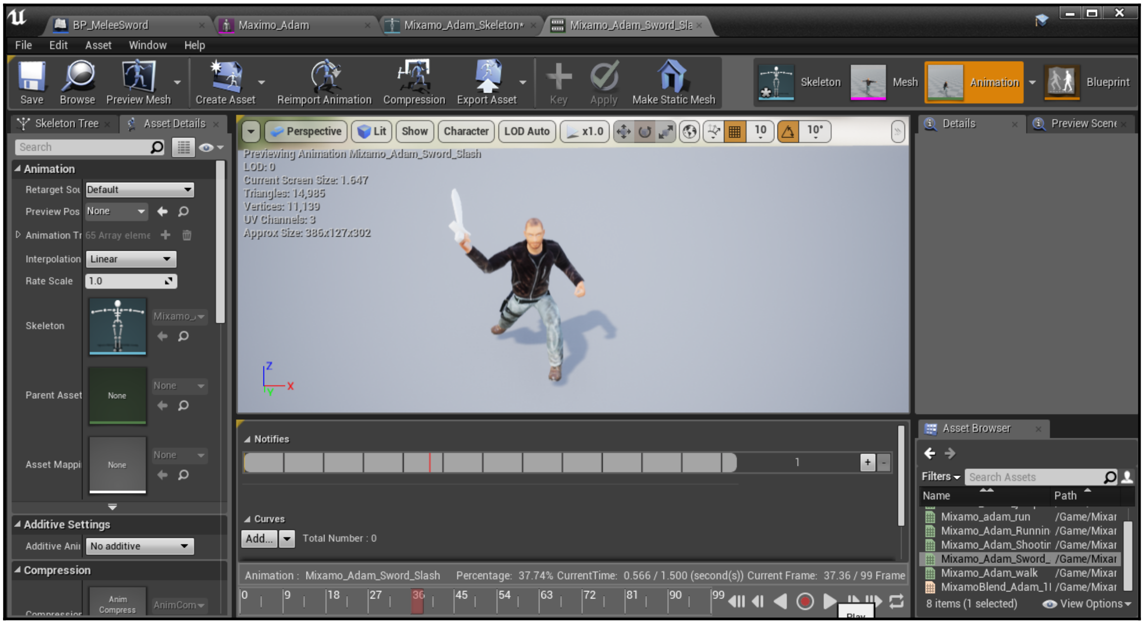
3. You should see Adam grip the sword in the image of the model, on the right-hand side of the following screenshot:

4. Now, click on **RightHandSocket** and zoom in on Adam's hand. We need to adjust the positioning of the socket in the preview so that the sword fits in it correctly.
5. Use the move and rotate manipulators or manually change the socket parameters in the Details window to line the sword up so that it fits in his hand correctly:

A real-world tip

If you have several sword models that you want to switch in and out of the same `RightHandSocket`, you will need to ensure quite a bit of uniformity (lack of anomalies) between the different swords that are supposed to go in that same socket.

6. You can preview your animations with the sword in the hand by going to the **Animation** tab in the top-right corner of the screen:



However, if you launch your game, Adam won't be holding a sword. That's because adding the sword to the socket in **Persona** is for preview purposes only.

Code to equip the player with a sword

To equip your player with a sword from the code and permanently bind it to the actor, instantiate an `AMeleeWeapon` instance and attach it to `RightHandSocket` after the monster instance is initialized. We do this in `PostInitializeComponents()` since, in this function, the `Mesh` object will have been fully initialized already.

In the `Monster.h` file, add a hook to select the `Blueprint` class name (`UClass`) of a melee weapon to use. Also, add a hook for a variable to actually store the `MeleeWeapon` instance using the following code:

```
// The MeleeWeapon class the monster uses
// If this is not set, he uses a melee attack
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
   MonsterProperties)
UClass* BPMeleeWeapon;
```

```
// The MeleeWeapon instance (set if the character is using
// a melee weapon)
AMeleeWeapon* MeleeWeapon;
```

Also, make sure that you add `#include "MeleeWeapon.h"` at the top of the file. Now, select the `BP_MeleeSword` blueprint in your monster's blueprint class.

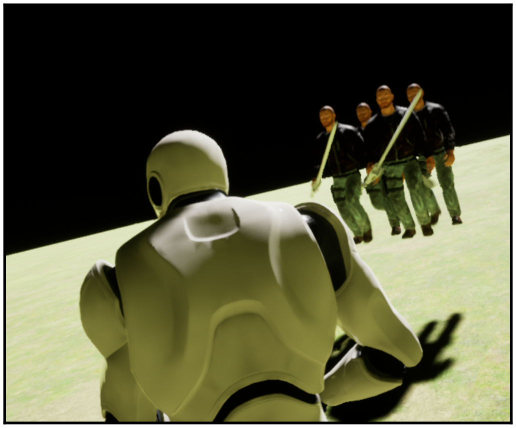
In the C++ code, you need to instantiate the weapon. To do so, we need to declare and implement a `PostInitializeComponents` function for the `Monster` class. In `Monster.h`, add a prototype declaration:

```
virtual void PostInitializeComponents() override;
```

`PostInitializeComponents` runs after the monster object's constructor has completed and all of the components of the object are initialized (including the blueprint construction). So, it is the perfect time to check whether the monster has a `MeleeWeapon` blueprint attached to it or not and to instantiate this weapon if it does. The following code is added to instantiate the weapon in the `Monster.cpp` implementation of `AMonster::PostInitializeComponents()`:

```
void AMonster::PostInitializeComponents()
{
    Super::PostInitializeComponents();

    // instantiate the melee weapon if a bp was selected
    if (BPMeleeWeapon)
    {
        MeleeWeapon = GetWorld()->SpawnActor<AMeleeWeapon>(
            BPMeleeWeapon, FVector(), FRotator());

        if (MeleeWeapon)
        {
            const USkeletalMeshSocket *socket = GetMesh()->GetSocketByName(
                FName("RightHandSocket")); // be sure to use correct
                                    // socket name!
            socket->AttachActor(MeleeWeapon, GetMesh());
            MeleeWeapon->WeaponHolder = this;
        }
    }
}
```

Also, make sure to put `#include "Engine/SkeletalMeshSocket.h"` at the top of the file. The monsters will now start with swords in hand if `BPMeleeWeapon` is selected for that monster's blueprint:



Triggering the attack animation

By default, there is no connection between our C++ `Monster` class and triggering the attack animation; in other words, the `MixamoAnimBP_Adam` class has no way of knowing when the monster is in the attack state.

Therefore, we need to update the animation blueprint of the Adam skeleton (`MixamoAnimBP_Adam`) to include a query in the `Monster` class variable listing and check whether the monster is in an attacking state. We haven't worked with animation blueprints (or blueprints in general) in this book before, but follow these instructions step by step and you should see it come together.

> **TIP** I'll introduce blueprints terminology gently here, but I'll encourage you to have a look at Zak Parrish's tutorial series at `https://www.youtube.com/playlist?list=PLZlv_N0_O1gbYMYfhhdzfW1tUV4jU0YxH` for your first introduction to blueprints.

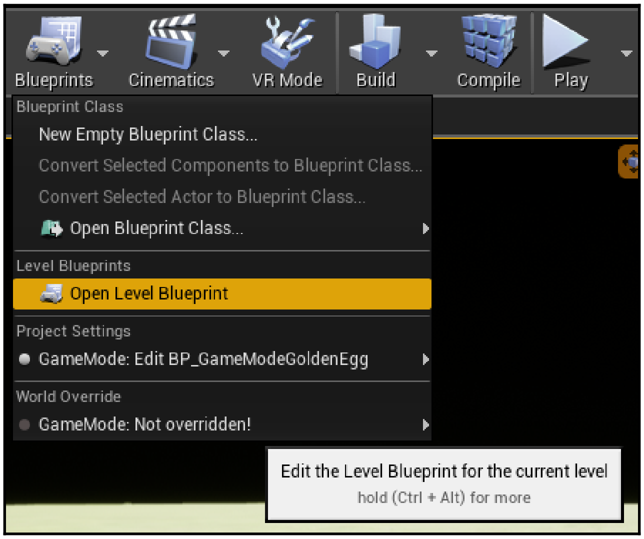
Blueprint basics

A UE4 blueprint is a visual realization of the code (not to be confused with how sometimes people say that a C++ class is a metaphorical blueprint of a class instance). In UE4 blueprints, instead of actually writing code, you drag and drop elements onto a graph and connect them to achieve the desired play. By connecting the right nodes to the right elements, you can program anything you want in your game.

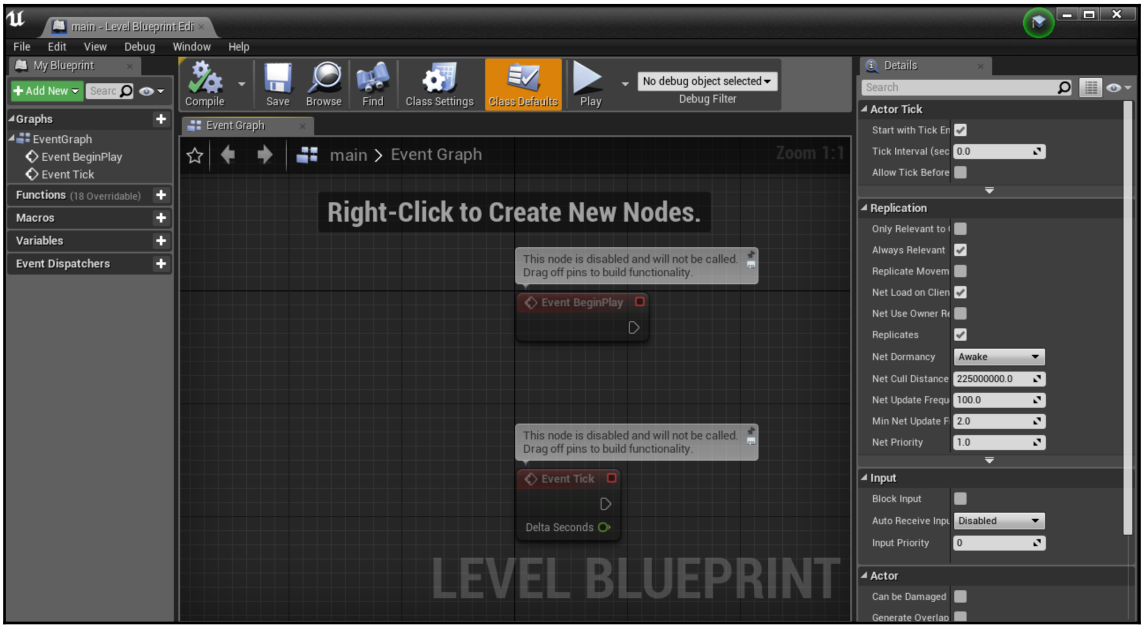
> TIP: This book does not encourage the use of blueprints since we are trying to encourage you to write your own code instead. Animations, however, are best worked with blueprints, because that is what artists and designers will know.

Let's start writing a sample blueprint to get a feel of how they work:

1. Click on the **Blueprints** menu bar at the top and select **Open Level Blueprint**, as shown in the following screenshot:

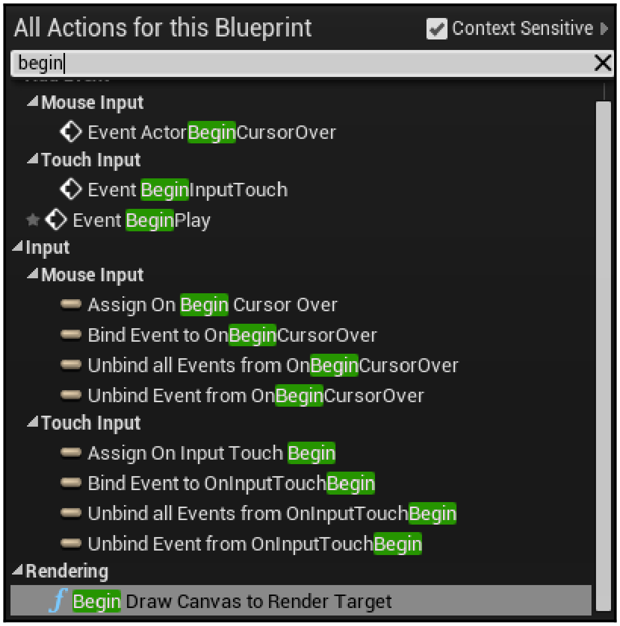


The **Level Blueprint** option executes automatically when you begin the level. Once you open this window, you should see a blank slate to create your gameplay on, as shown here:

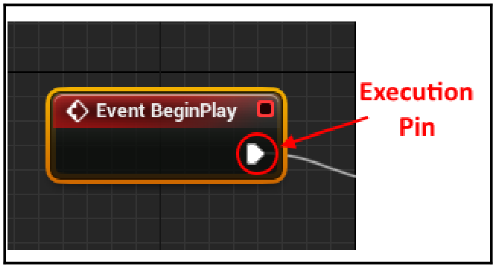


2. Right-click anywhere on the graph paper.
3. Start typing `begin` and click on the **Event Begin Play** option from the drop-down list that appears.

> **TIP**
>
> Ensure that the **Context Sensitive** checkbox is checked, as shown in the following screenshot:



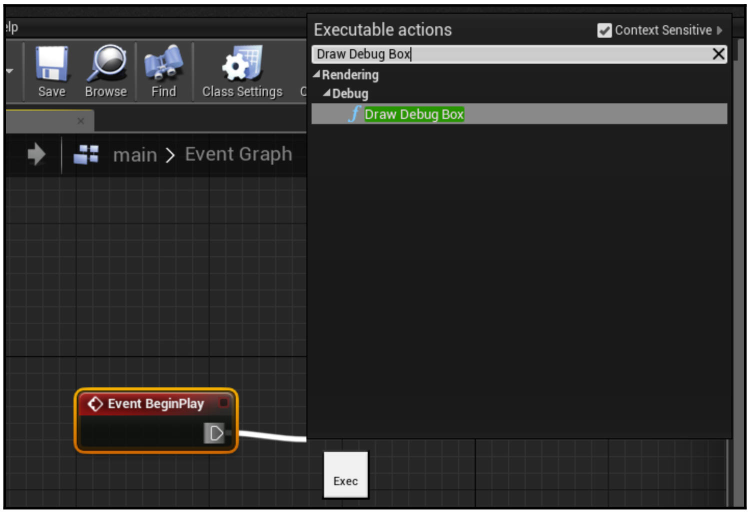
4. Immediately after you click on the **Event Begin Play** option, a red box will appear on your screen. It has a single white pin on the right-hand side. This is called the execution pin, as shown here:



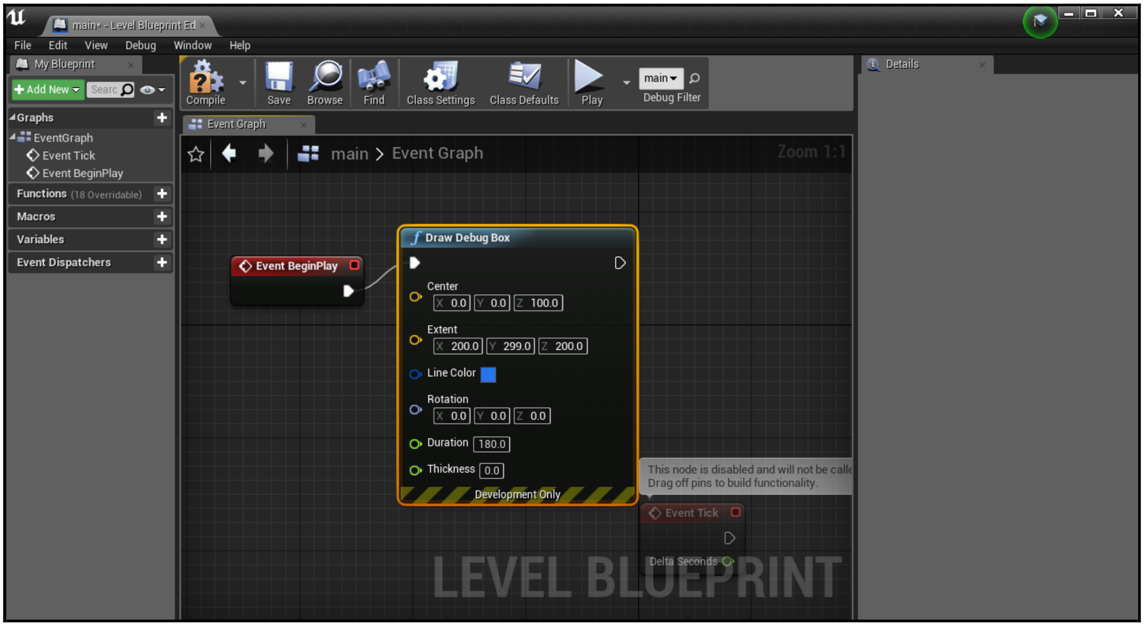
The first thing that you need to know about animation blueprints is the white pin execution path (the white line). If you've seen a blueprint graph before, you must have noticed a white line going through the graph, as shown in the following diagram:

The white pin execution path is pretty much equivalent to having lines of code lined up and run one after the other. The white line determines which nodes will get executed and in what order. If a node does not have a white execution pin attached to it, then that node will not get executed at all.

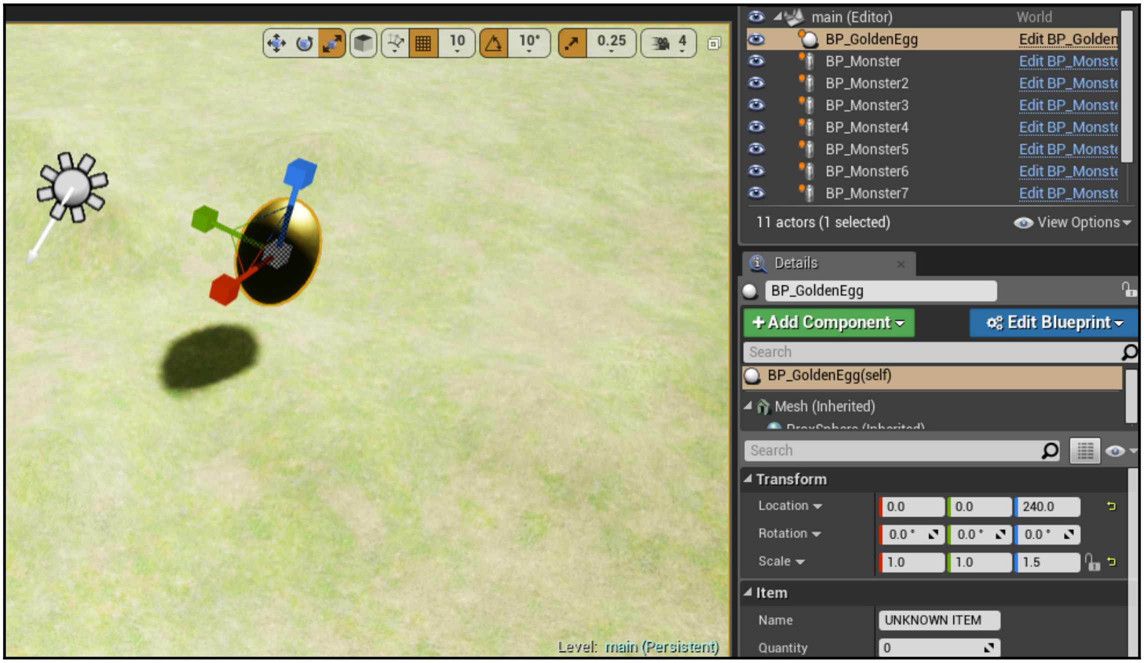
5. Drag the white execution pin off **Event Begin Play**.
6. Start by typing `draw debug box` in the **Executable actions** dialog.
7. Select the first thing that pops up (**fDraw Debug Box**), as shown here:



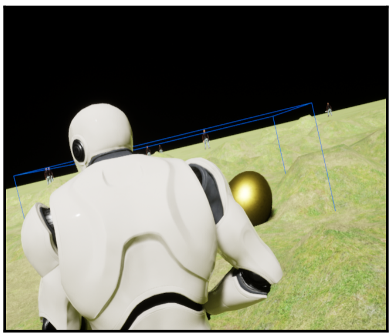
8. Fill in some details for how you want the box to look. Here, I selected the color blue for the box, the center of the box at (0, 0, 100), the size of the box to be (200, 200, 200), and a duration of 180 seconds (be sure to enter a duration that is long enough that you can see the result), as shown in the following screenshot:



9. Now, click on the **Play** button to realize the graph. Remember that you have to find the world's origin to see the debug box.
10. Find the world's origin by placing a golden egg at (`0, 0`, (some `z` value)), as shown in the following screenshot, or try increasing the line thickness to make it more visible:



This is how the box will look in the level:



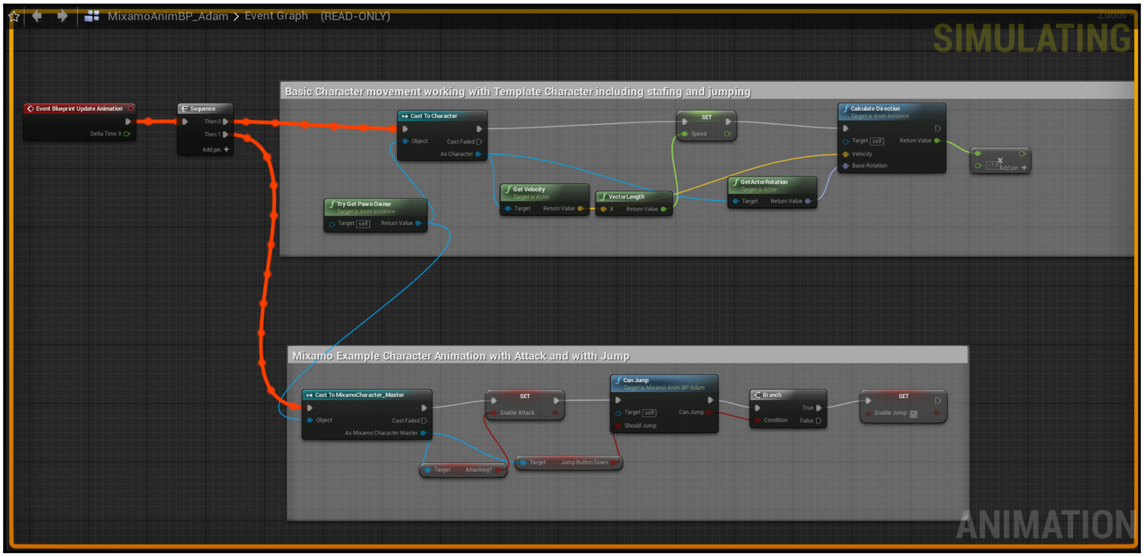
Modifying the animation blueprint for Mixamo Adam

To integrate our attack animation, we have to modify the blueprint. Under **Content Browser**, open up `MixamoAnimBP_Adam`.

The first thing you'll notice is that the graph has two sections above the **Event Notifies** section:

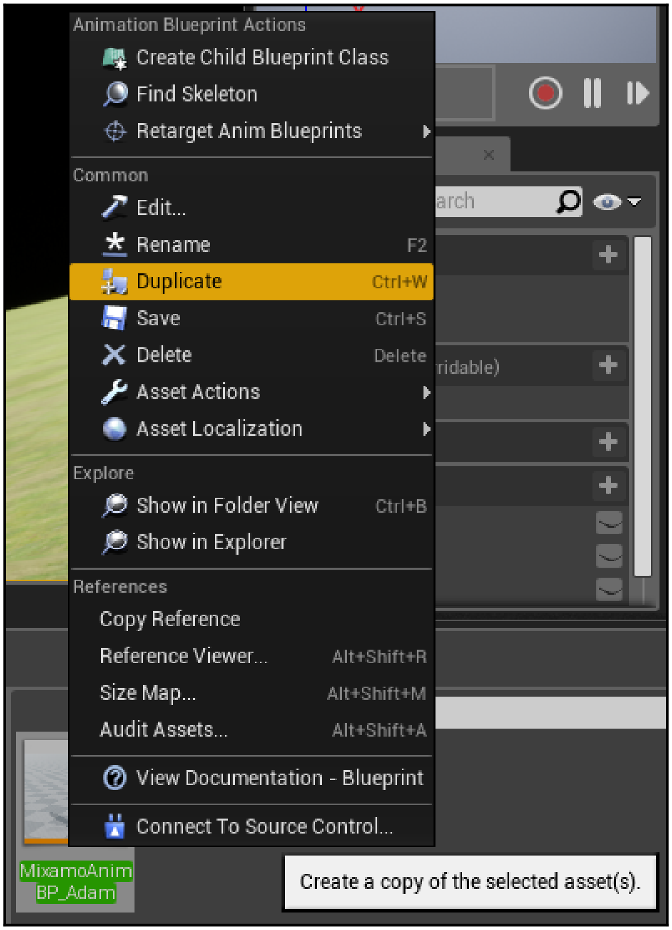
- The top section is marked as **Basic Character movement**....
- The bottom section says **Mixamo Example Character Animation**....

Basic character movement is in charge of the walking and running movements of the model. We'll be working in the **Mixamo Example Character Animation with Attack and Jump** section, which is responsible for the attack animation. We'll be working in the latter section of the graph, as shown in the following screenshot:



When you first open the graph, it starts out by zooming in on a section near the bottom. To scroll up, right-click the mouse and drag it upwards. You can also zoom out using the mouse wheel or by holding down the *Alt* key and the right mouse button while moving the mouse up.

Before proceeding, you might want to duplicate the **MixamoAnimBP_Adam** resource so that you don't damage the original, in case you need to go back and change something later. This allows you to easily go back and correct things if you find that you've made a mistake in one of your modifications, without having to reinstall a fresh copy of the whole animation package into your project:

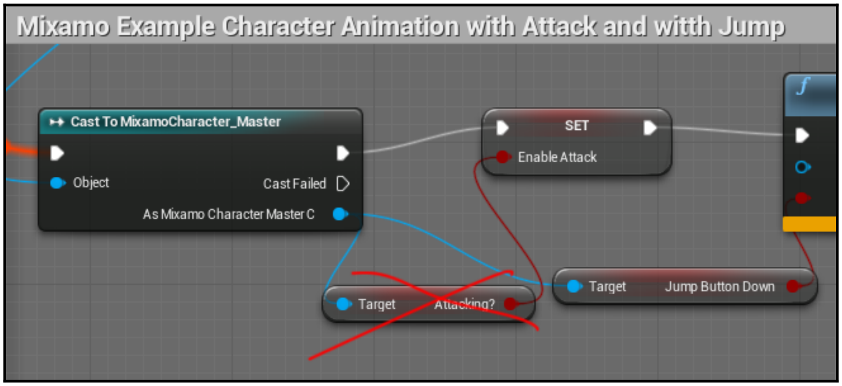


TIP

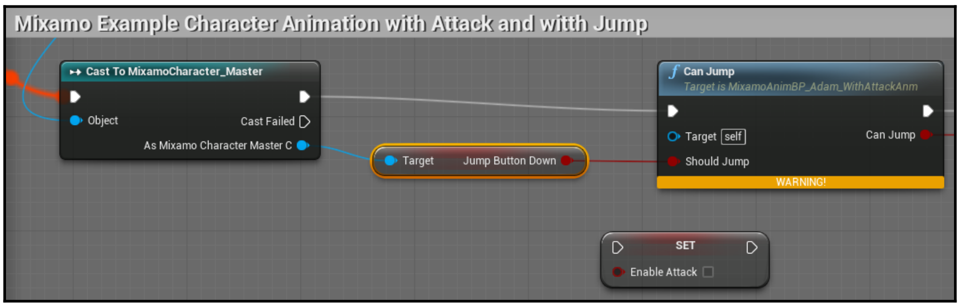
When assets are added to a project from the Unreal Launcher, a copy of the original asset is made, so you can modify **MixamoAnimBP_Adam** in your project now and get a fresh copy of the original assets in a new project later.

We're going to do only a few things to make Adam swing the sword when he is attacking. Let's do it in this order:

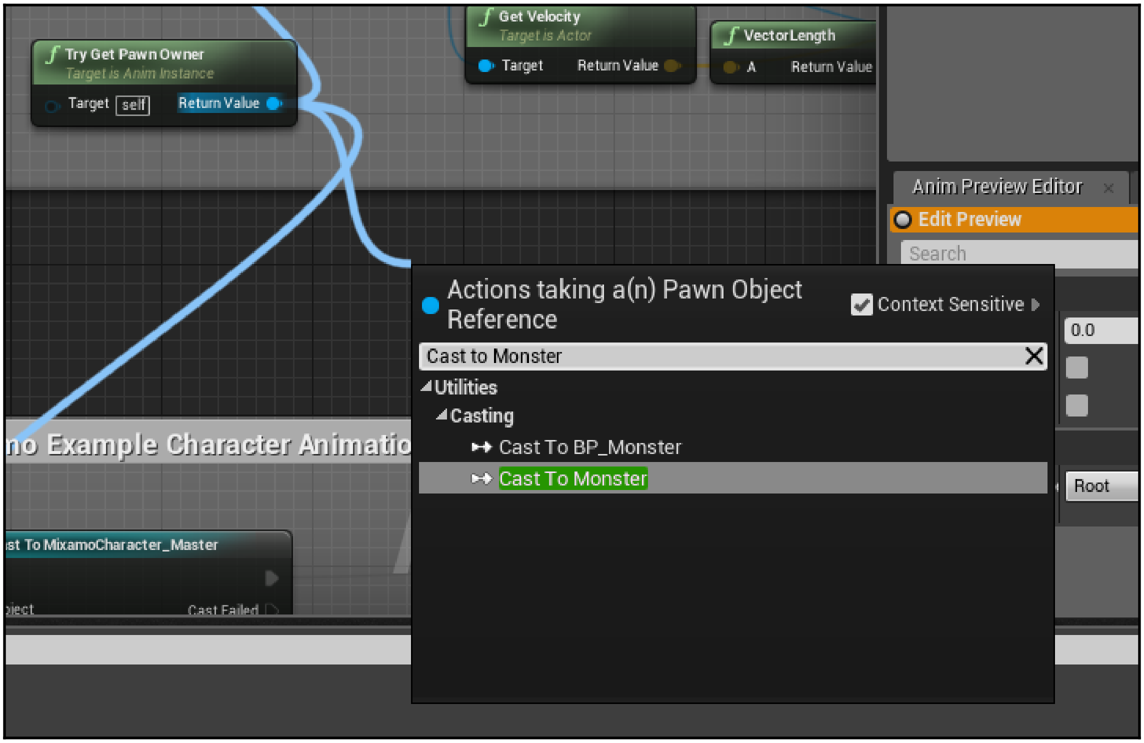
1. Delete the node that says **Attacking?**:



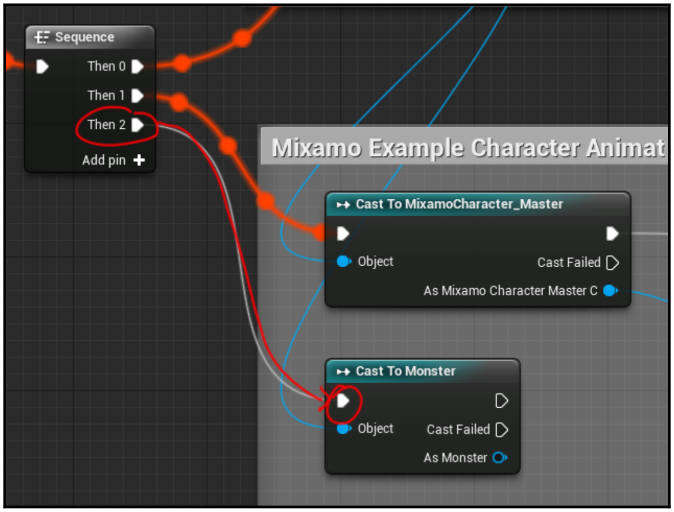
2. Rearrange the nodes, as follows, with the **Enable Attack** node by itself at the bottom:



3. We're going to handle the monster that this animation is animating. Scroll up the graph a bit and drag the blue dot marked as **Return Value** in the **Try Get Pawn Owner** dialog. Drop it into your graph and, when the pop-up menu appears, select **Cast to Monster** (ensure that **Context Sensitive** is checked, or the **Cast to Monster** option will not appear). The **Try Get Pawn Owner** option gets the `Monster` instance that owns the animation, which is just the `AMonster` class object, as shown in the following screenshot:

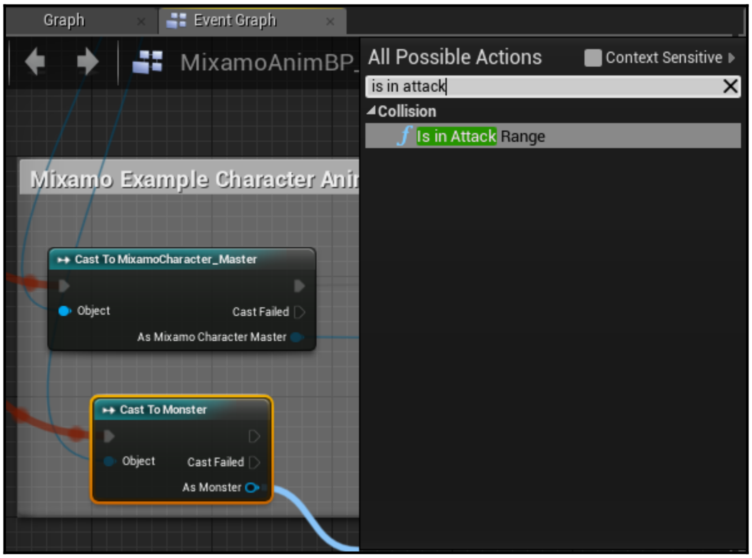


4. Click on **+** in the **Sequence** dialog and drag another execution pin from the **Sequence** group to the **Cast to Monster** node instance, as shown in the following screenshot. This ensures that the **Cast to Monster** instance actually gets executed:

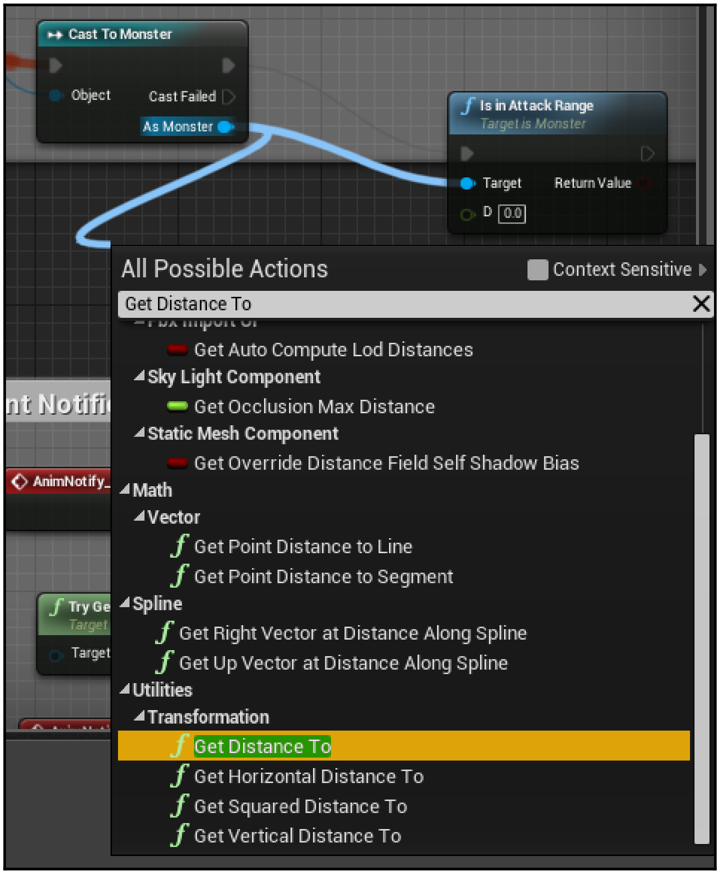


5. The next step is to pull out the pin from the **As Monster** terminal of the **Cast to Monster** node and look for the **Is in Attack Range** property:

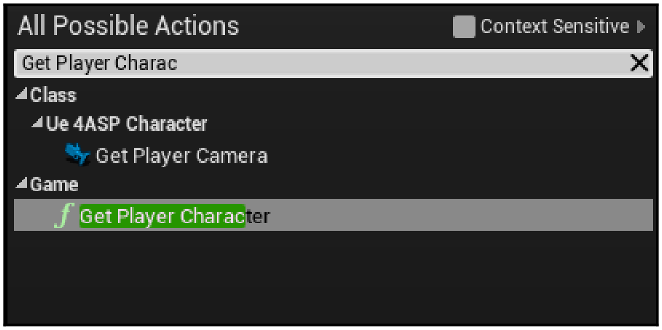
> **TIP**
>
> For this to show up, you need to go back to `Monster.h` and add the following line before the **is in Attack Range** function and compile the project (this will be explained a little later):
> `UFUNCTION(BlueprintCallable, Category = Collision)`



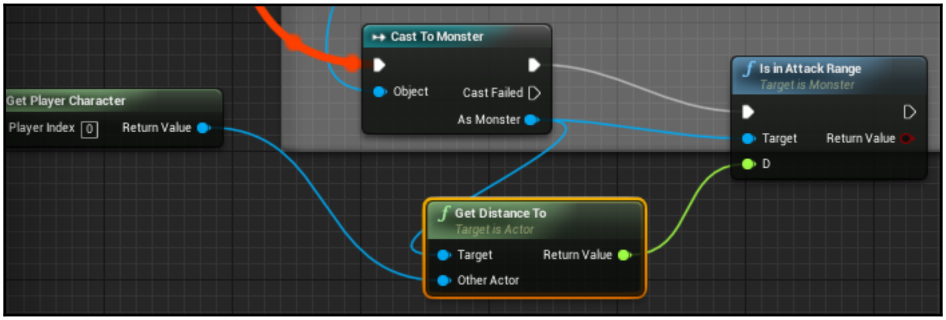
6. There should automatically be a line from the white execution pin from the **Cast to Monster** node on the left-hand side of the **Is in Attack Range** node on the right-hand side. Next, drag another line from **As Monster** and this time look for **Get Distance To**:



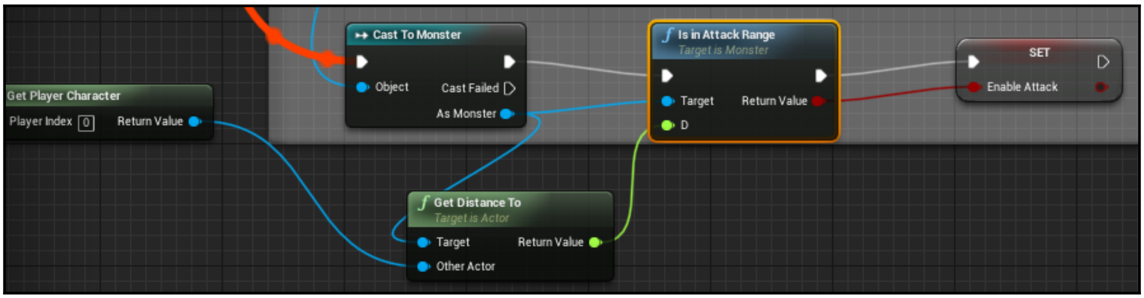
7. You need to add a node to get the **Player** character to send to the **Other Actor** node of **Get Distance To**. Just right-click anywhere and look for **Get Player Character**:



8. Connect the **Return Value** node from **Get Player Character** to **Other Actor**, and **Return Value** from **Get Distance To** to **D** in **Is In Attack Range**:



9. Pull the white and red pins over to the **SET** node, as shown here:



> **TIP**
>
> The equivalent pseudocode of the preceding blueprint is something similar to the following:
>
> ```
> if(   Monster.isInAttackRangeOfPlayer() )
> {
>     Monster.Animation = The Attack Animation;
> }
> ```

Test your animation. The monster should swing only when it's within the player's range. If it doesn't work and you created a duplicate, make sure that you switched the `animBP` to the duplicate. Also, the default animation is shooting, not swinging a sword. We will be fixing that later.

Code to swing the sword

We want to add an animation notify event when the sword is swung:

1. Declare and add a blueprint callable C++ function to your `Monster` class:

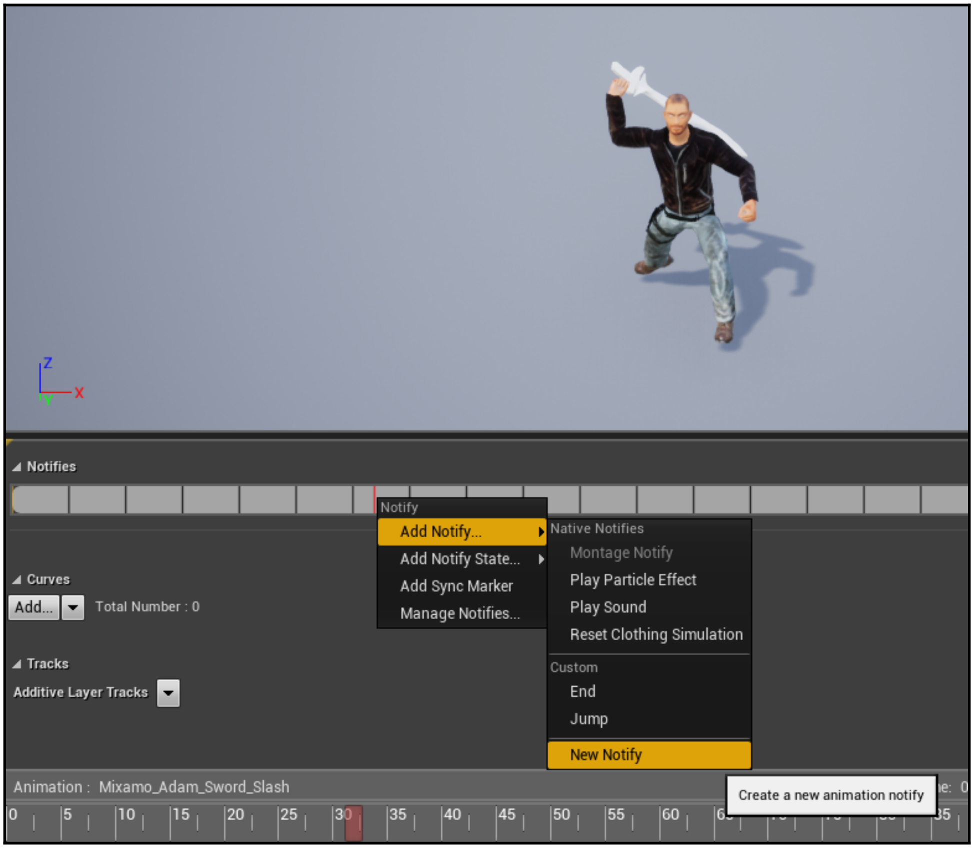
   ```
   // in Monster.h:
   UFUNCTION( BlueprintCallable, Category = Collision )
   void SwordSwung();
   ```

   The `BlueprintCallable` statement means that it will be possible to call this function from blueprints. In other words, `SwordSwung()` will be a C++ function that we can invoke from a blueprints node, as shown here:

   ```
   // in Monster.cpp
   void AMonster::SwordSwung()
   {
     if( MeleeWeapon )
     {
       MeleeWeapon->Swing();
     }
   }
   ```

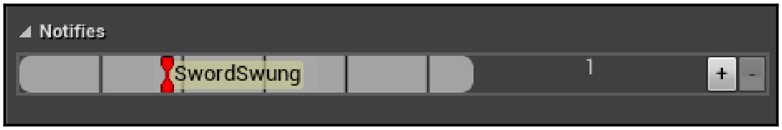
2. Open the **Mixamo_Adam_Sword_Slash** animation by double-clicking on it from your **Content Browser** (it should be in **MixamoAnimPack/Mixamo_Adam/Anims/Mixamo_Adam_Sword_Slash**).
3. Find the point where Adam starts swinging his sword.

4. Right-click that point on the **Notifies** bar and select **New Notify** under **Add Notify...**, as shown in the following screenshot:



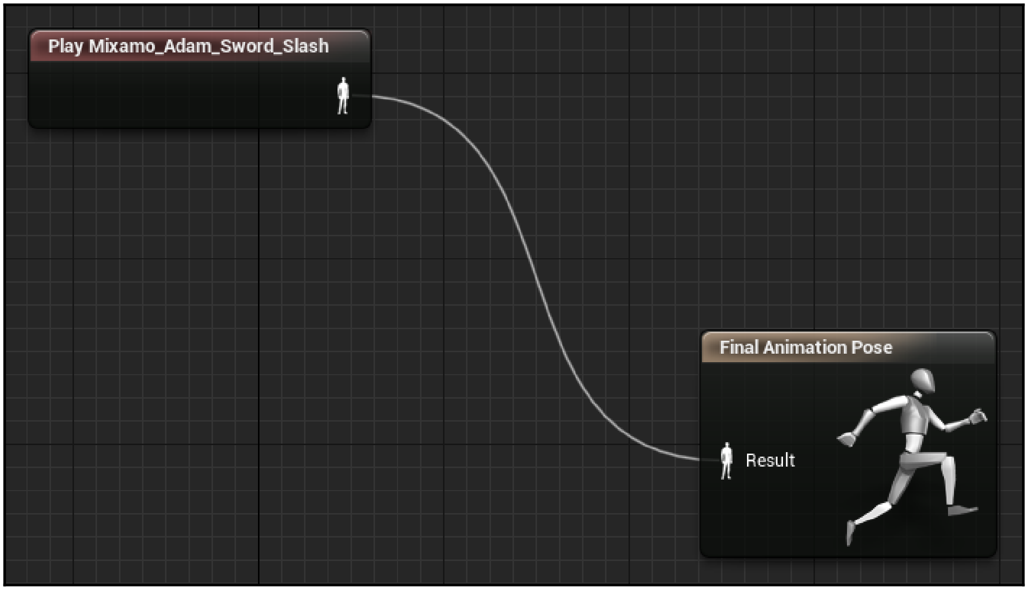
5. Name the notification `SwordSwung`:

The notification name should appear in your animation's timeline, as follows:



6. Save the animation and then open up your version of **MixamoAnimBP_Adam** again.
7. Underneath the **SET** group of nodes, create the following graph:

8. The **AnimNotify_SwordSwung** node appears when you right-click in the graph (with **Context Sensitive** turned on) and start typing `SwordSwung`. The **Cast To Monster** node is again fed in from the **Try Get Pawn Owner** node, as in step 2 of the *Modifying the animation blueprint for Mixamo Adam* section.
9. **Sword Swung** is our blueprint-callable C++ function in the `AMonster` class (you will need to compile the project for this to show up).
10. You also need to go into the **AnimGraph** tab of **MaximoAnimBP_Adam**.
11. Double-click on **State Machine** to open that graph.
12. Double-click on the **attacking** state to open that.
13. Select the one on the left that says **Play Mixamo_Adam Shooting**.
14. Shooting is the default animation, but that's clearly not what we want to happen. So, delete that and right-click and look for **Play Mixamo_Adam_Sword_Slash**. Then, click from the little icon of a person and drag it the **Result** of **Final Animation Pose**:



If you start the game now, your monsters will execute their attack animation whenever they actually attack. If you also override `TakeDamage` in the `AAvatar` class to reduce HP when the sword's bounding box comes into contact with you, you will see your HP bar go down a bit (recall that the HP bar was added at the end of `Chapter 8`, *Actors and Pawns*, as an exercise):

Projectile or ranged attacks

Ranged attacks usually involve a projectile of some sort. Projectiles are things such as bullets, but they can also include things such as lightning magic attacks or fireball attacks. To program a projectile attack, you should spawn a new object and only apply the damage to the player if the projectile reaches the player.

To implement a basic bullet in UE4, we should derive a new object type. I derived an `ABullet` class from the `AActor` class, as shown in the following code:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Components/SphereComponent.h"
#include "Bullet.generated.h"

UCLASS()
class GOLDENEGG_API ABullet : public AActor
{
 GENERATED_BODY()

public:
 // Sets default values for this actor's properties
 ABullet(const FObjectInitializer& ObjectInitializer);

 // How much damage the bullet does.
 UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
 Properties)
 float Damage;

 // The visible Mesh for the component, so we can see
 // the shooting object
 UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
 Collision)
 UStaticMeshComponent* Mesh;

 // the sphere you collide with to do impact damage
 UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
 Collision)
 USphereComponent* ProxSphere;

 UFUNCTION(BlueprintNativeEvent, Category = Collision)
 void Prox(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp,
 int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
```

```
 // You shouldn't need this unless you get a compiler error that it can't
find this function.
 virtual int Prox_Implementation(UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor, UPrimitiveComponent* OtherComp,
 int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult); };
```

The `ABullet` class has a couple of important members in it, as follows:

- A `float` variable for the damage that a bullet does on contact
- A `Mesh` variable for the body of the bullet
- A `ProxSphere` variable to detect when the bullet finally hits something
- A function to be run when `Prox` is detected near an object

The constructor for the `ABullet` class should have the initialization of the `Mesh` and `ProxSphere` variables. In the constructor, we set `RootComponent` as the `Mesh` variable and then attached the `ProxSphere` variable to the `Mesh` variable. The `ProxSphere` variable will be used for collision checking. Collision checking for the `Mesh` variable should be turned off, as shown in the following code:

```
ABullet::ABullet(const FObjectInitializer& ObjectInitializer)
    : Super(ObjectInitializer)
{
    Mesh =
ObjectInitializer.CreateDefaultSubobject<UStaticMeshComponent>(this,
        TEXT("Mesh"));
    RootComponent = Mesh;

    ProxSphere =
ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this,
        TEXT("ProxSphere"));
    ProxSphere->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepWorldTransform);

    ProxSphere->OnComponentBeginOverlap.AddDynamic(this,
        &ABullet::Prox);
    Damage = 1;
}
```

We initialized the `Damage` variable to `1` in the constructor, but this can be changed in the UE4 editor once we create a blueprint out of the `ABullet` class. Next, the `ABullet::Prox_Implementation()` function should deal damage to the actor if we collide with the other actor's `RootComponent`. We can implement this by code:

```
int ABullet::Prox_Implementation(UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if (OtherComp != OtherActor->GetRootComponent())
    {
        // don't collide w/ anything other than
        // the actor's root component
        return -1;
    }

    OtherActor->TakeDamage(Damage, FDamageEvent(), NULL, this);
    Destroy();
    return 0;
}
```

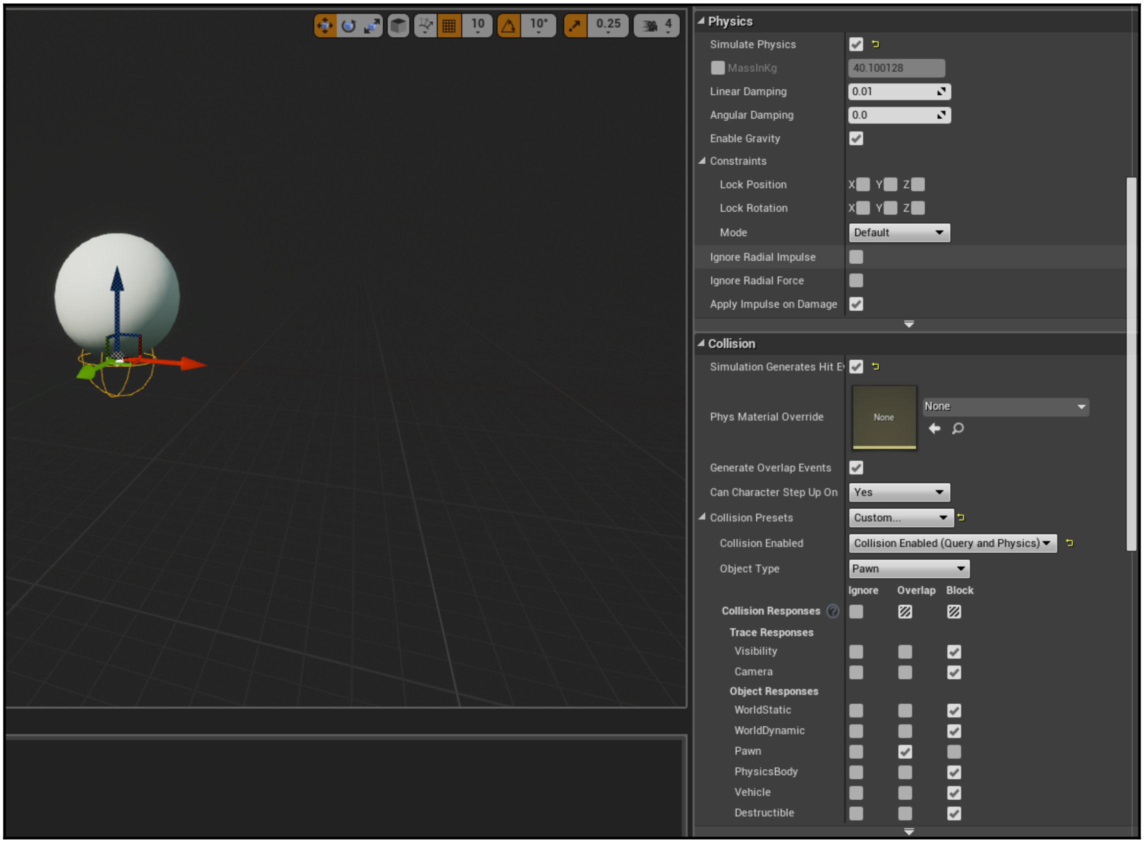
Bullet physics

To make bullets fly through the level, you can use UE4's physics engine.

Create a blueprint based on the `ABullet` class. I selected **Shape_Sphere** for the mesh and scaled it down to a more appropriate size. The bullet's mesh should have collision physics enabled, but the bullet's bounding sphere will be used to calculate damage.

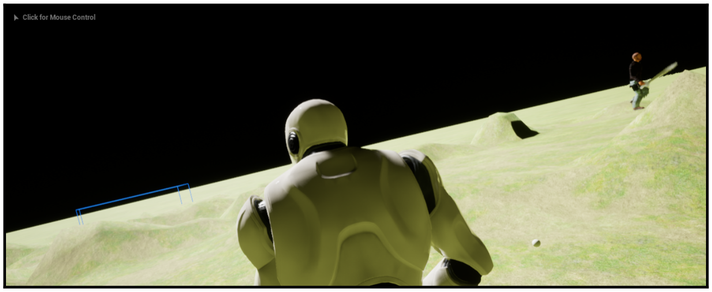
Configuring the bullet to behave properly is mildly tricky, so we'll cover this in four steps, as follows:

1. Select **Mesh (Inherited)** in the **Components** tab. The `ProxSphere` variable should be under the **Mesh**.
2. In the **Details** tab, check both **Simulate Physics** and **Simulation Generates Hit Events**.
3. From the **Collision Presets** drop-down list, select **Custom...**.
4. Select **Collision Enabled (Query and Physics)** from the **Collision Enabled** dropdown. Also, check the **Collision Responses** boxes, as shown; check **Block** for most types (**WorldStatic**, **WorldDynamic**, and so on) and check **Overlap**, but only for **Pawn**:



The **Simulate Physics** checkbox makes the `ProxSphere` property experience gravity and the impulse forces exerted on it. An impulse is a momentary thrust of force, which we'll use to drive the shot of the bullet. If you do not check the **Simulation Generate Hit Events** checkbox, then the ball will drop to the floor. What **Blocking All Collision** does is ensure that the ball can't pass through anything.

If you drag and drop a couple of these `BP_Bullet` objects from the **Content Browser** tab directly into the world now, they will simply fall to the floor. You can kick them once they are on the floor. The following screenshot shows the ball object on the floor:



However, we don't want our bullets falling on the floor. We want them to be shot. So, let's put our bullets in the `Monster` class.

Adding bullets to the monster class

Let's look at a step-by-step way to do this:

1. Add a member to the `Monster` class that receives a blueprint instance reference. That's what the `UClass` object type is for. Also, add a blueprint configurable `float` property to adjust the force that shoots the bullet, as shown in the following code:

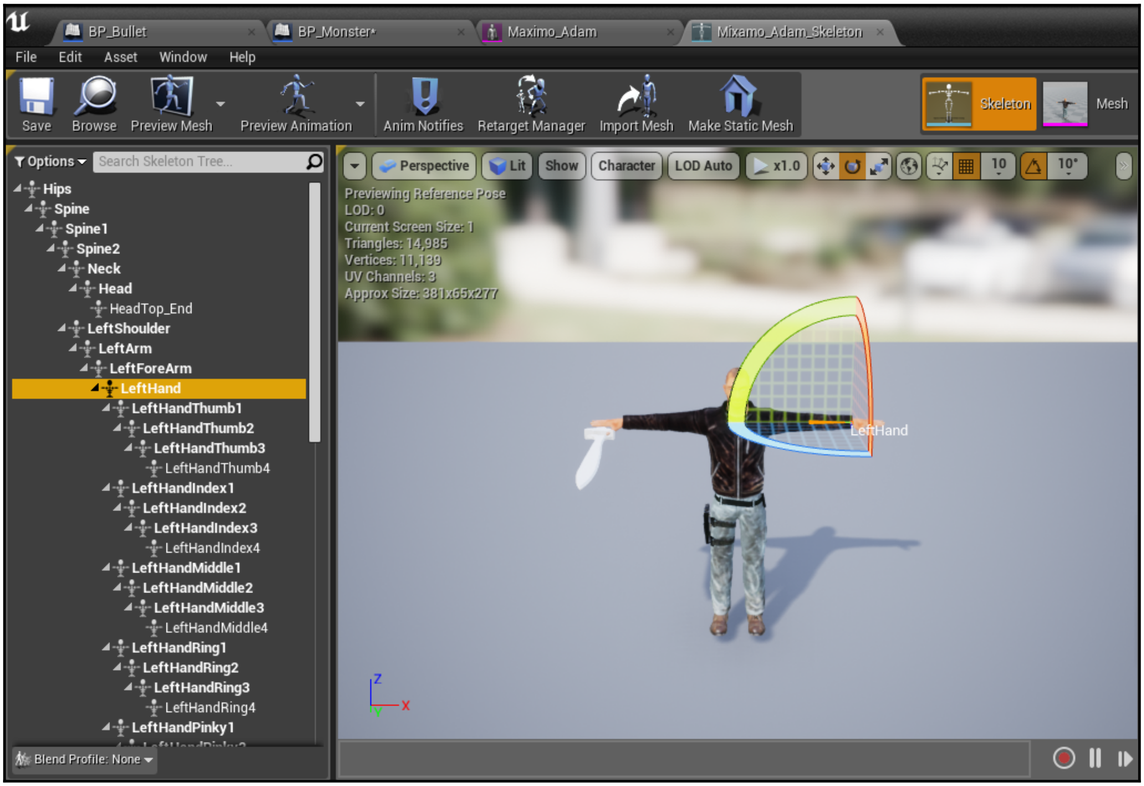
```
// The blueprint of the bullet class the monster uses
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
   MonsterProperties)
UClass* BPBullet;
// Thrust behind bullet launches
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
   MonsterProperties)
float BulletLaunchImpulse;
```

2. Compile and run the C++ project and open your `BP_Monster` blueprint.
3. You can now select a blueprint class under `BPBullet`, as shown in the following screenshot:

4. Once you've selected a blueprint class type to instantiate when the monster shoots, you have to program the monster to shoot when the player is in its range.

   Where does the monster shoot from? Actually, it should shoot from a bone. If you're not familiar with this terminology, bones are just reference points in the model's mesh. A model mesh is usually made up of many "bones."

5. To see some bones, open up the **Mixamo_Adam** mesh by double-clicking on the asset in the **Content Browser** tab, as shown in the following screenshot:



6. Go to the **Skeleton** tab and you will see all the monster's bones in a tree view list in the left-hand side. What we want to do is select a bone from which bullets will be emitted. Here, I've selected the `LeftHand` option.

TIP: An artist will normally insert an additional bone into the model mesh to emit the particle, which is likely to be on the tip of the nozzle of a gun.

Working from the base model mesh, we can get the `Mesh` bone's location and have the monster emit the `Bullet` instances from that bone in the code.

The complete monster `Tick` and `Attack` functions can be obtained using the following code:

```
void AMonster::Tick(float DeltaSeconds)
{
  Super::Tick( DeltaSeconds );

  // move the monster towards the player
  AAvatar *avatar = Cast<AAvatar>(
   UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
  if( !avatar ) return;

  FVector playerPos = avatar->GetActorLocation();
  FVector toPlayer = playerPos - GetActorLocation();
  float distanceToPlayer = toPlayer.Size();

  // If the player is not the SightSphere of the monster,
  // go back
  if( distanceToPlayer > SightSphere->GetScaledSphereRadius() )
  {
    // If the player is OS, then the enemy cannot chase
    return;
  }

  toPlayer /= distanceToPlayer;  // normalizes the vector

  // At least face the target
  // Gets you the rotator to turn something
  // that looks in the `toPlayer` direction
  FRotator toPlayerRotation = toPlayer.Rotation();
  toPlayerRotation.Pitch = 0; // 0 off the pitch
  RootComponent->SetWorldRotation( toPlayerRotation );

  if( isInAttackRange(distanceToPlayer) )
  {
    // Perform the attack
    if( !TimeSinceLastStrike )
    {
      Attack(avatar);
```

```
      }

      TimeSinceLastStrike += DeltaSeconds;
      if( TimeSinceLastStrike > AttackTimeout )
      {
        TimeSinceLastStrike = 0;
      }

      return;  // nothing else to do
    }
    else
    {
      // not in attack range, so walk towards player
      AddMovementInput(toPlayer, Speed*DeltaSeconds);
    }
  }
```

The `AMonster::Attack` function is relatively simple. Of course, we first need to add a prototype declaration in the `Monster.h` file in order to write our function in the `.cpp` file:

```
void Attack(AActor* thing);
```

In `Monster.cpp`, we implement the `Attack` function, as follows:

```
void AMonster::Attack(AActor* thing)
{
  if( MeleeWeapon )
  {
    // code for the melee weapon swing, if
    // a melee weapon is used
    MeleeWeapon->Swing();
  }
  else if( BPBullet )
  {
    // If a blueprint for a bullet to use was assigned,
    // then use that. Note we wouldn't execute this code
    // bullet firing code if a MeleeWeapon was equipped
    FVector fwd = GetActorForwardVector();
    FVector nozzle = GetMesh()->GetBoneLocation( "RightHand" );
    nozzle += fwd * 155;// move it fwd of the monster so it
     doesn't
    // collide with the monster model
    FVector toOpponent = thing->GetActorLocation() - nozzle;
    toOpponent.Normalize();
    ABullet *bullet = GetWorld()->SpawnActor<ABullet>(
     BPBullet, nozzle, RootComponent->GetComponentRotation());

    if( bullet )
```

```
    {
      bullet->Firer = this;
      bullet->ProxSphere->AddImpulse(
        toOpponent*BulletLaunchImpulse );
    }
    else
    {
      GEngine->AddOnScreenDebugMessage( 0, 5.f,
      FColor::Yellow, "monster: no bullet actor could be spawned.
       is the bullet overlapping something?" );
    }
  }
}
```

Also, make sure that you add `#include "Bullet.h"` at the top of the file. We leave the code that implements the melee attack as it is. Assuming that the monster is not holding a melee weapon, we then check whether the `BPBullet` member is set. If the `BPBullet` member is set, it means that the monster will create and fire an instance of the `BPBullet` blueprinted class.

Pay special attention to the following line:

```
ABullet *bullet = GetWorld()->SpawnActor<ABullet>(BPBullet,
   nozzle, RootComponent->GetComponentRotation() );
```

This is how we add a new actor to the world. The `SpawnActor()` function puts an instance of `UCLASS` that you pass in `spawnLoc`, with some initial orientation.

After we spawn the bullet, we call the `AddImpulse()` function on its `ProxSphere` variable to rocket it forward.

Also, add the following line to **Bullet.h**:

```
AMonster *Firer;
```

Player knockback

To add a knockback to the player, I added a member variable to the `Avatar` class called `knockback`. A knockback happens whenever the avatar gets hurt:

```
FVector knockback; // in class AAvatar
```

To figure out the direction to knock the player back when he gets hit, we need to add some code to `AAvatar::TakeDamage`. This overrides the version in the `AActor` class, so first, add this to **Avatar.h**:

```
virtual float TakeDamage(float DamageAmount, struct FDamageEvent const&
DamageEvent, class AController* EventInstigator, AActor* DamageCauser)
override;
```

Compute the direction vector from the attacker toward the player and store this vector in the `knockback` variable:

```
float AAvatar::TakeDamage(float DamageAmount, struct FDamageEvent const&
DamageEvent, class AController* EventInstigator, AActor* DamageCauser)
{
    // add some knockback that gets applied over a few frames
    knockback = GetActorLocation() - DamageCauser->GetActorLocation();
    knockback.Normalize();
    knockback *= DamageAmount * 500; // knockback proportional to damage
    return AActor::TakeDamage(DamageAmount, DamageEvent, EventInstigator,
DamageCauser);
}
```

In `AAvatar::Tick`, we apply the knockback to the avatar's position:

```
void AAvatar::Tick( float DeltaSeconds )
{
  Super::Tick( DeltaSeconds );

  // apply knockback vector
  AddMovementInput( -1*knockback, 1.f );

  // half the size of the knockback each frame
  knockback *= 0.5f;
}
```

Since the knockback vector reduces in size with each frame, it becomes weaker over time, unless the knockback vector gets renewed with another hit.

TIP

For the bullets to work, you need to set **BPMelee Weapon** to **None**. You should also increase the size of **AttackRangeSphere** and adjust **Bullet Launch Impulse** to a value that works.

Summary

In this chapter, we explored how to instantiate monsters on the screen that run after the player and attack him. We used different spheres to detect whether the monster was in sight or attack range, and added the ability to have either melee or shooting attacks, depending on whether or not the monster has a melee weapon. If you want to experiment further, you can try changing animations for shooting, or add an extra sphere and have the monster fire while still moving and switch to melee when in attack range. In the next chapter, we'll expand the capabilities of the monsters further by looking into advanced Artificial Intelligence techniques.

12
Building Smarter Monsters with Advanced AI

The monsters we have so far don't really do a lot. They stand in one place until they get in range to see you, and then they head over to you and either do a melee attack or a shooting attack, depending on what you have set up. In a real game, you want your characters to do a lot more than that so they seem more real. That's where **Artificial Intelligence** (**AI**) comes in.

AI is a huge topic that has entire books devoted to it, but we'll be covering a few ways UE4 supports making AI programming easier so you can easily create more realistic monsters. We'll be doing a quick overview of the following topics:

- Navigation - Pathfinding and the NavMesh
- Behavior Trees
- Environment Query Systems
- Flocking
- Machine Learning and Neural Networks
- Genetic Algorithms

If you're interested in learning more after all that, there are many great books you can look at for a more in-depth look at what else you can do with AI.

Navigation – pathfinding and the NavMesh

Right now, the monsters we've created only move in one direction—in a straight line directly toward your position. But what if there are mountains, buildings, trees, rivers, or other objects in the way? In many cases, a straight line just isn't possible. Right now, if the monster runs into a wall, it'll just stay there, which isn't very realistic. This is where pathfinding comes in.

What is pathfinding?

Pathfinding is a way to figure out a path (usually the shortest and/or easiest one) to a destination. Picture the entire environment as a grid, with numbers in each cell saying how difficult it is to navigate. So a cell with a wall blocking the way would have a very high value, and a steep path could have a higher value than an easy path. The goal of pathfinding is to find the path with the lowest overall value when you add up all the cells along that path.

There are different algorithms, or methods, for handling pathfinding available. The most well-known one is called A* (pronounced *A star*).

What is A*?

We won't be using A* here, but you should at least be familiar with it if you plan on doing AI programming in the future, so I'll do a brief overview. A* basically searches the cells surrounding the character, prioritizing the ones with the lowest cost. It calculates the cost of the path so far (by adding up the costs up until that point) plus a heuristic, a guess on the cost from that point to the goal.

There are many ways of calculating a heuristic. It could be something as simple as the distance directly to the goal (as the crow flies, you might say). It's better for the results if the heuristic is actually lower than what the actual cost will turn out to be, so that works well.

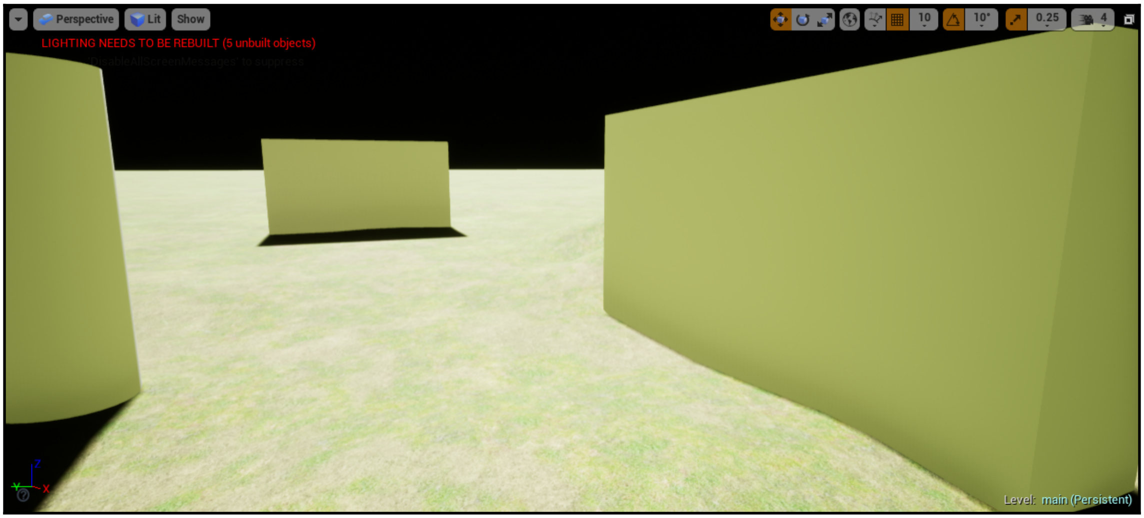
Once you find the cell with the lowest cost, then go one step further and look at the cells surrounding that cell. You continue until you reach the goal. If you find yourself at a cell you've been to before and the total path cost this way is lower, you can replace it with the lower-cost path. This helps you get a shorter path. Once you get to the goal, you can follow the path backward and you'll have a complete path to the goal.

You can find much more information on A* and other pathfinding algorithms online or in books on AI. You will need to know them if you do this in more complex projects, but for this, UE4 has a much simpler and easier way: using a `NavMesh`.

Using a NavMesh

A `NavMesh` is an object in UE4 that you can place in your world to tell it what parts of the environment you want characters to be able to navigate. To do this, perform the following steps:

1. Add some obstacles. You can add cubes, cylinders, or anything else you want to add to block movement, like this:



2. Once you've set up the level the way you want it, in the **Modes** window, go to **Volumes**, find **Nav Mesh Bounds Volume**, drag it onto the level, and scale it to cover the entire area you want the monsters to be able to navigate.

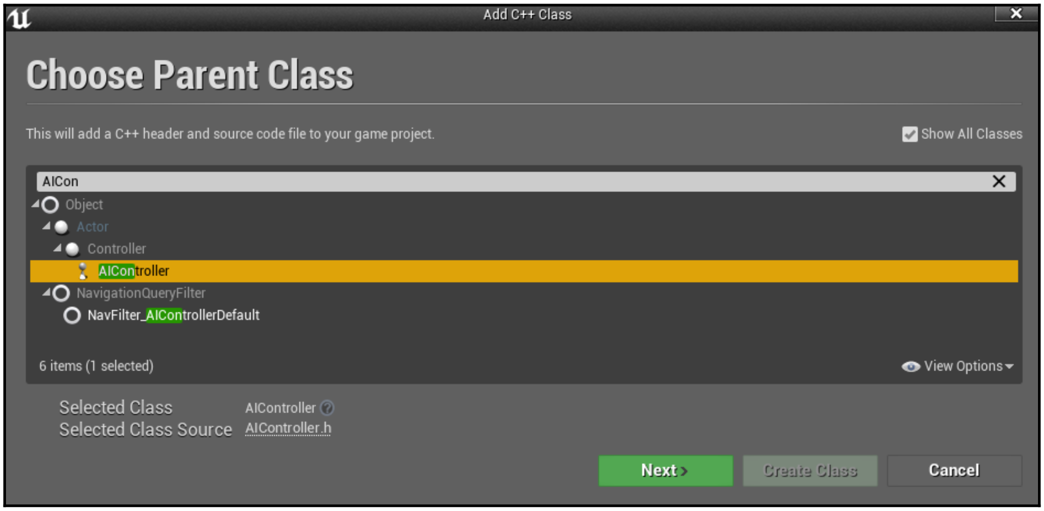
TIP

If you try it now, you'll still see the monsters walk into walls and just stop. That's because we need to change the way movement is handled. We'll do this by creating our own `AIController` class.

Creating an AIController class

Let's follow a step-by-step procedure to do this:

1. Create a new C++ class. In this case, you'll need to check the **Show All Classes** checkbox and search to find `AIController`:



2. Name the class `MonsterAIController`. Your `MonsterAIController.h` should look like this:

```
UCLASS()
class GOLDENEGG_API AMonsterAIController : public AAIController
{
    GENERATED_BODY()
public:
    //Start following the player
    void StartFollowingPlayer();
};
```

`MonsterAIController.cpp` should implement that function as follows:

```
void AMonsterAIController::StartFollowingPlayer()
{
    AActor *player = Cast<AActor>(
        UGameplayStatics::GetPlayerPawn(GetWorld(), 0));
    FVector playerPos = player->GetActorLocation();
    MoveToLocation(playerPos);
}
```

Also make sure to add `#include "Kismet/GameplayStatics.h"` at the top of the file.

3. Go back into the `Tick()` function in `Monster.cpp`. Find the following line in the `else` clause:

```
AddMovementInput(toPlayer, Speed*DeltaSeconds);
```

Delete this line and replace it with this:

```
if (GetController() != nullptr)
{
    Cast<AMonsterAIController>(GetController())-
    >StartFollowingPlayer();
}
```

Also add `#include "MonsterAIController.h"` at the top of the file, and go into `BP_Monster` and change the **Ai Controller** class to `MonsterAIController`. Now the monsters can find their way around the walls to you. If they don't move, check to make sure the `NavMesh` covers the area and is tall enough to cover the characters.

Behavior Tree

Right now, all the logic for controlling your monsters is in the `Tick()` function in `Monster.cpp`. But what you've done so far is pretty simple. In large, complex games, the monsters will have a lot more behaviors. They could patrol an area until they see you, or even communicate with you and only attack if the conversation doesn't go well. The logic for all this would get much too complicated to keep everything in one function, or even in the `AMonster` class.

Fortunately, UE4 has another way of managing complex tasks, and that is a Behavior Tree. A Behavior Tree lets you visually set up a series of tasks to make them easier to manage. Since we're focused on C++ here, we'll be creating the tasks themselves that way, but the overall tree seems to be easier to manage in **Blueprints**.

Behavior Trees are primarily controlled by two different types of nodes:

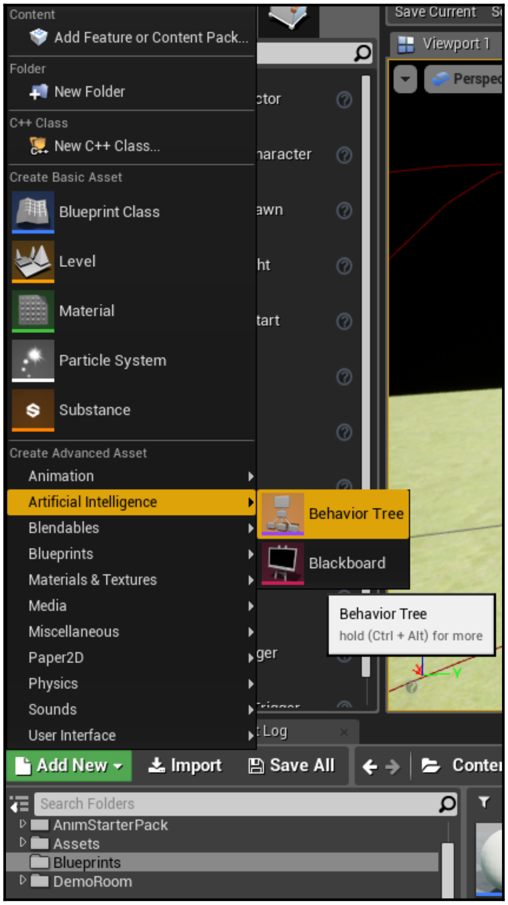
- **Selectors**: Selector will run through its children, from left to right, until one succeeds, and then goes back up the tree. Think of it like an `or` statement—once it finds one true argument, the `or` itself is true so it is done.

- **Sequences**: Sequences instead go through the children from left to right until one fails. This is more like an `and` statement that keeps going until something comes up as false, rendering the whole statement false.

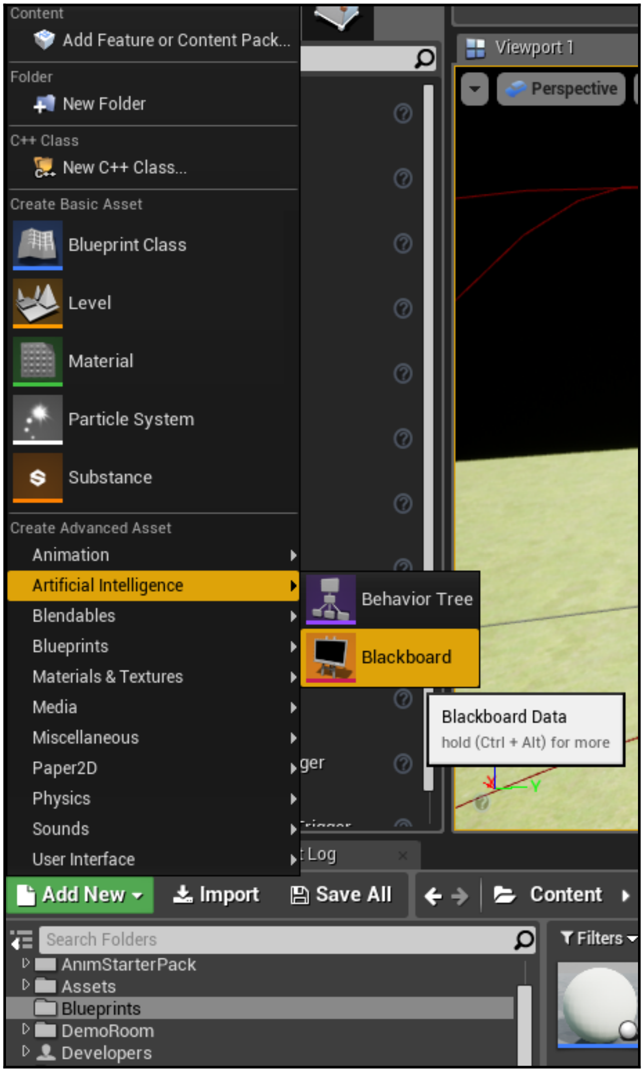
So if you want to run multiple steps, you will use Sequences, whereas if you just want to run one successfully and stop, you will use **Selector**.

Setting up the Behavior Tree

First, you need to go into your library (put it in a folder name that makes sense so you'll remember where to find it, or Blueprints will work) and, from **Add New**, select **Artificial Intelligence** | **Behavior Tree**:

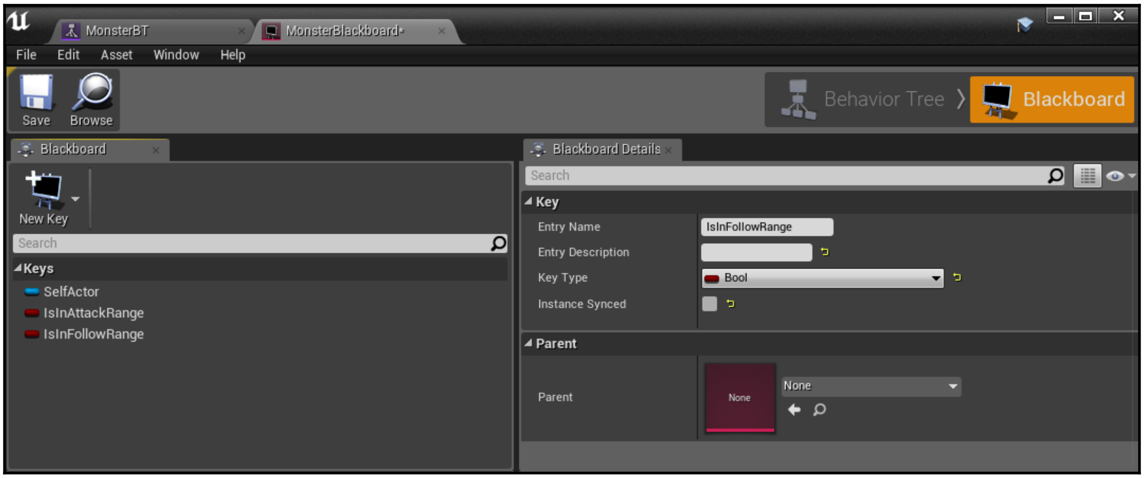


I named mine `MonsterBT`. You will also need to create a **Blackboard**. This stores the data you will use in the Behavior Tree and lets you transfer it easily between the AI Controller and the Behavior Tree. You create that by going to **Add New** and this time selecting **Artificial Intelligence** | **Blackboard**. I named this one `MonsterBlackboard`:



Setting up Blackboard values

Next, you'll want to set up values in the **Blackboard** you just created. You do this by selecting **New Key** and then selecting a type (in this case, **Bool**). For this, I've added two of them, **IsInAttackRange** and **IsInFollowRange**:

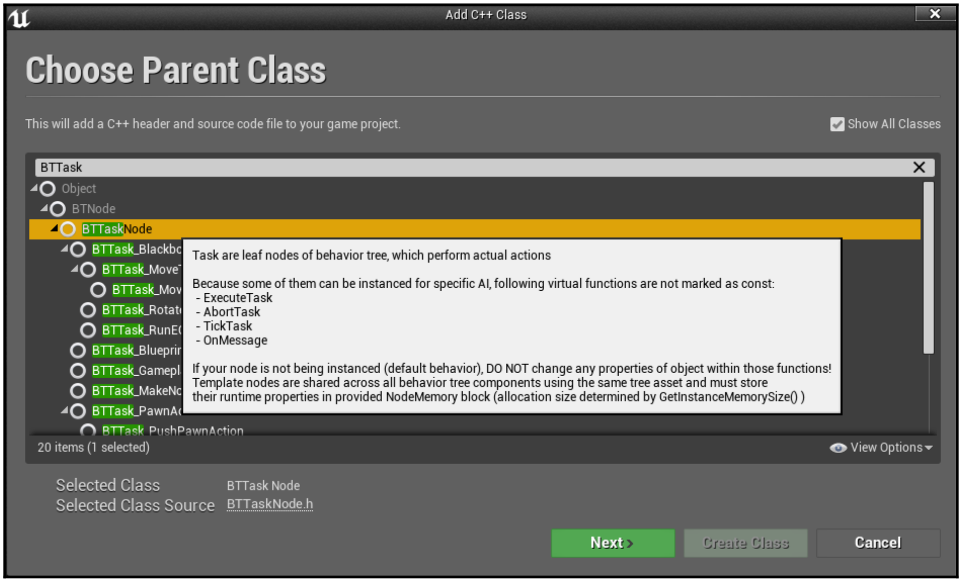


You can also give each one a description of what it is used for.

Setting up a BTTask

We will be creating a C++ task to handle following the player. To do this, perform the following steps:

1. Add a new C++ class and base it off **BTTaskNode** (you will need to view all classes and search for it):



I named the new class `BTTask_FollowPlayer`

2. In `BTTaskFollowPlayer.h`, add the following:

```
UCLASS()
class GOLDENEGG_API UBTTask_FollowPlayer : public UBTTaskNode
{
    GENERATED_BODY()
    virtual EBTNodeResult::Type ExecuteTask(UBehaviorTreeComponent&
OwnerComp, uint8* NodeMemory) override;
    virtual void OnGameplayTaskActivated(UGameplayTask& Task)
override {}
};
```

We won't be using `OnGameplayTaskActivated`, but, without declaring it, your code may not compile (if you get a complaint about it not being there, that's why)

3. In `BTTaskFollowPlayer.cpp`, add the following:

```
#include "BTTask_FollowPlayer.h"
#include "MonsterAIController.h"


EBTNodeResult::Type
UBTTask_FollowPlayer::ExecuteTask(UBehaviorTreeComponent&
OwnerComp, uint8* NodeMemory)
```

```
{
    AMonsterAIController* Controller =
Cast<AMonsterAIController>(OwnerComp.GetAIOwner());
    if (Controller == nullptr)
    {
        return EBTNodeResult::Failed;
    }

    Controller->StartFollowingPlayer();

    return EBTNodeResult::Succeeded;
}
```

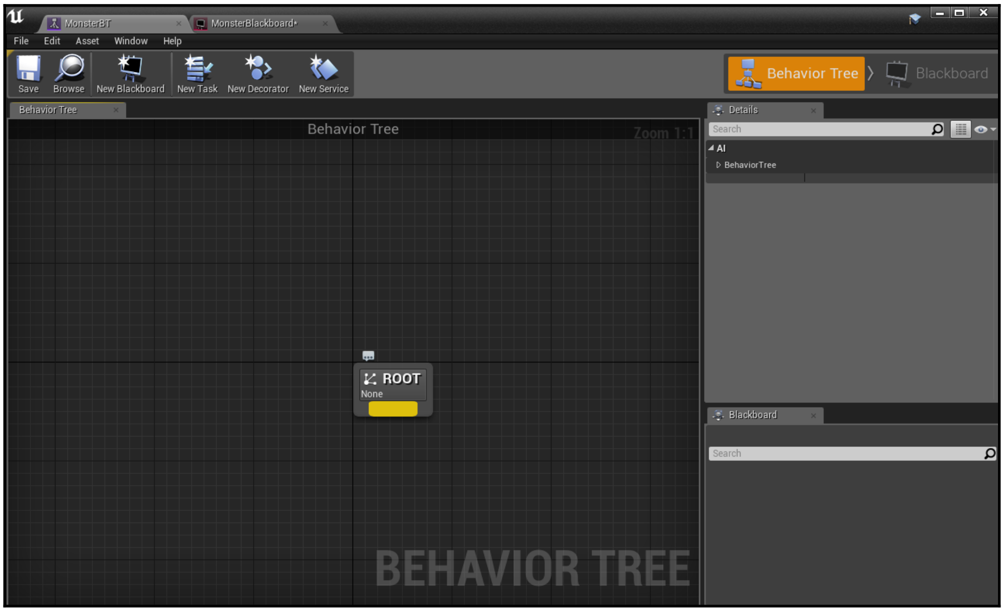
TIP

Once you have this working, you can go back and create another `BTTask` to handle attacking too, as well as any other behaviors you might want.

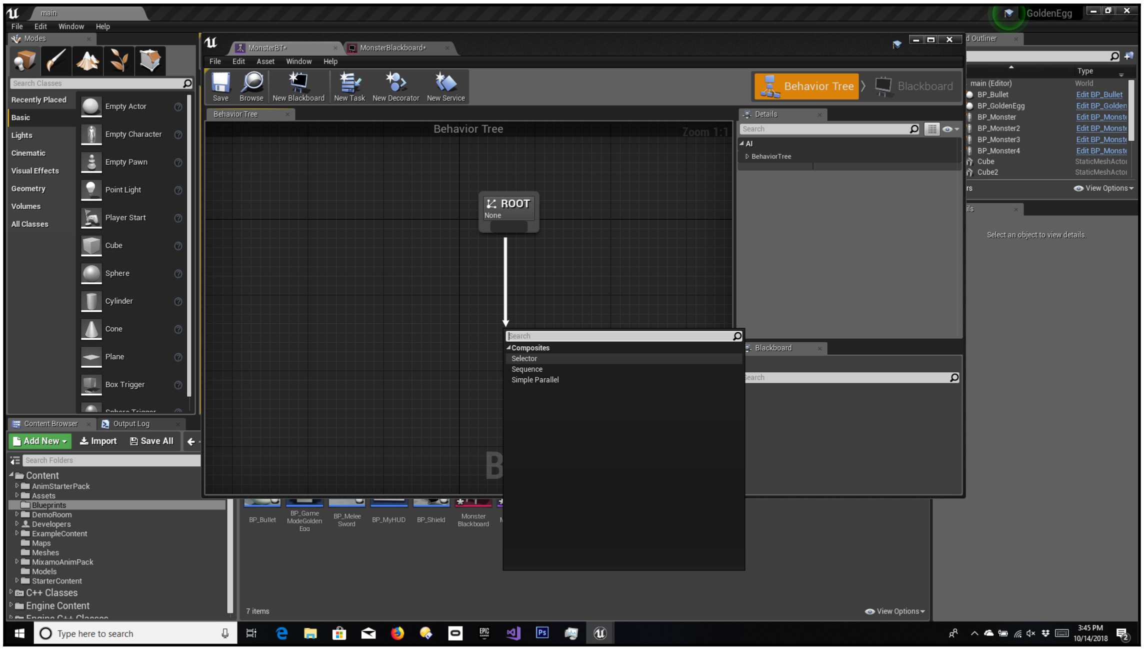
Setting Up the Behavior Tree itself

Once you have the task set up, it's time to set up the tree itself:

1. Double-click on it to open the **Blueprints**:

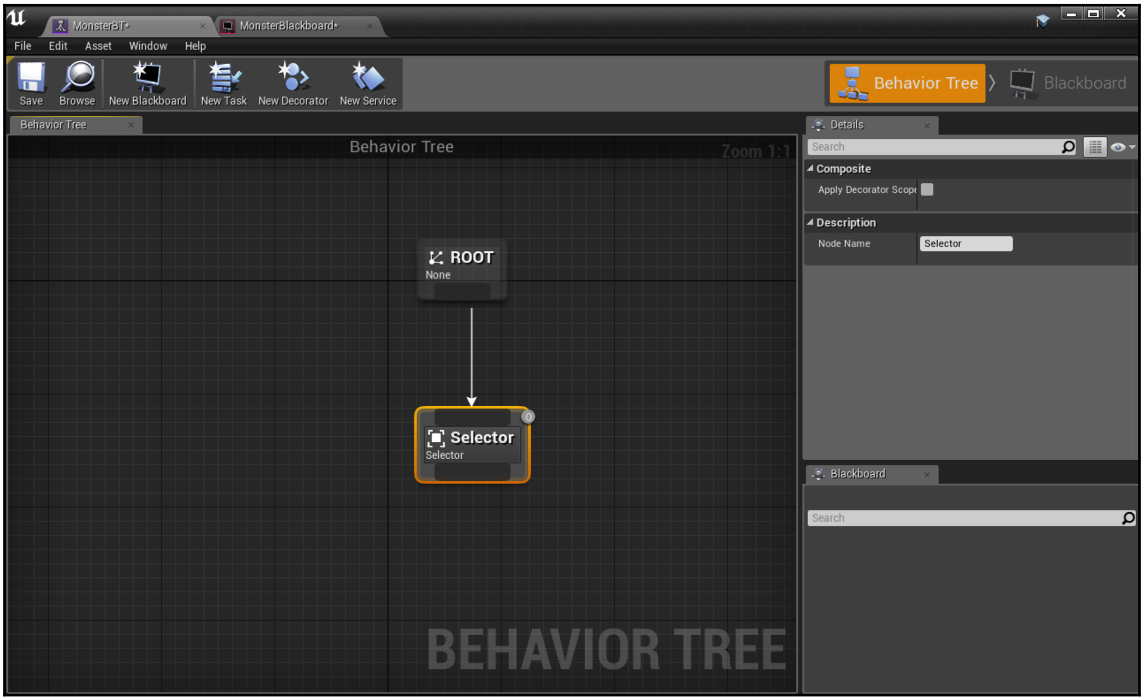


2. Click on the yellow area on the bottom of **Root** and drag it out to create a new node (it's black but turns yellow when the mouse rolls over it).
3. Select the type from the menu that comes up (we'll use **Selector**):



The selector icon in the center tab

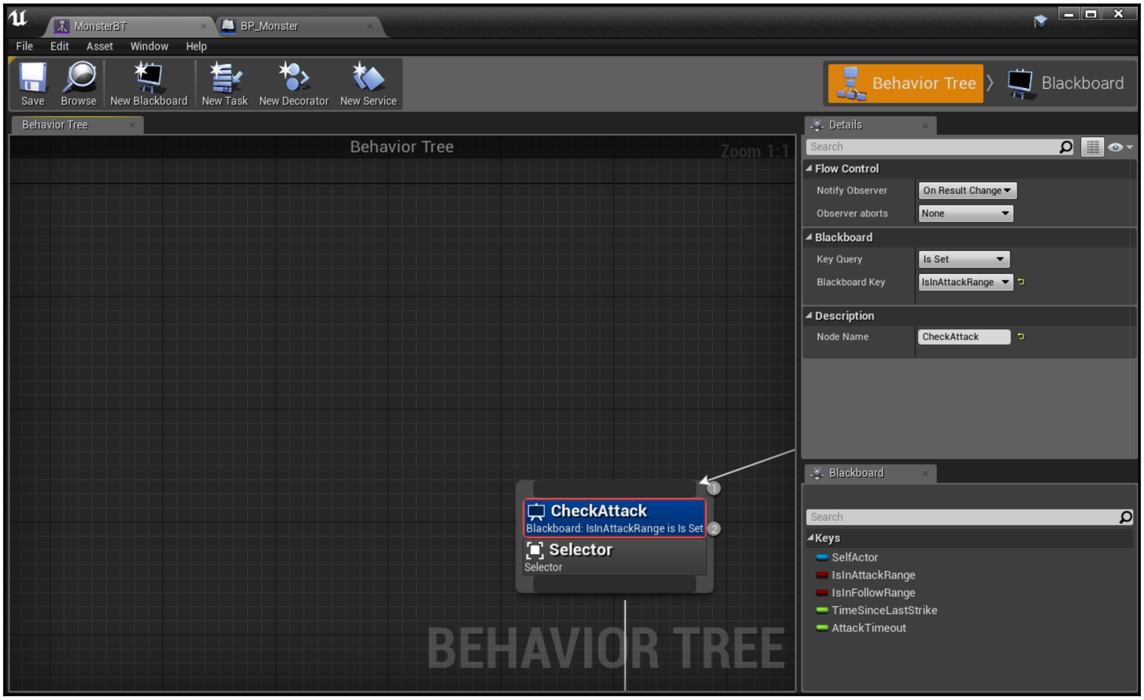
4. You should have the following:



As mentioned before, the **Selector** will go through nodes in left-to-right order until one succeeds, and then stop. In this case, we have three possible states: in attack range, in sight range, and neither (ignore the player). First, you want to check whether you're close enough to attack, which means you'll want to check **IsInAttackRange** in your **Blackboard**.

Don't do follow first because the attack range is still technically in follow range, but you don't want to use the follow functionality, so the **Selector** will stop after checking for follow range because that's the first check it makes, so it will never check for attack range (which is what it should really be checking).

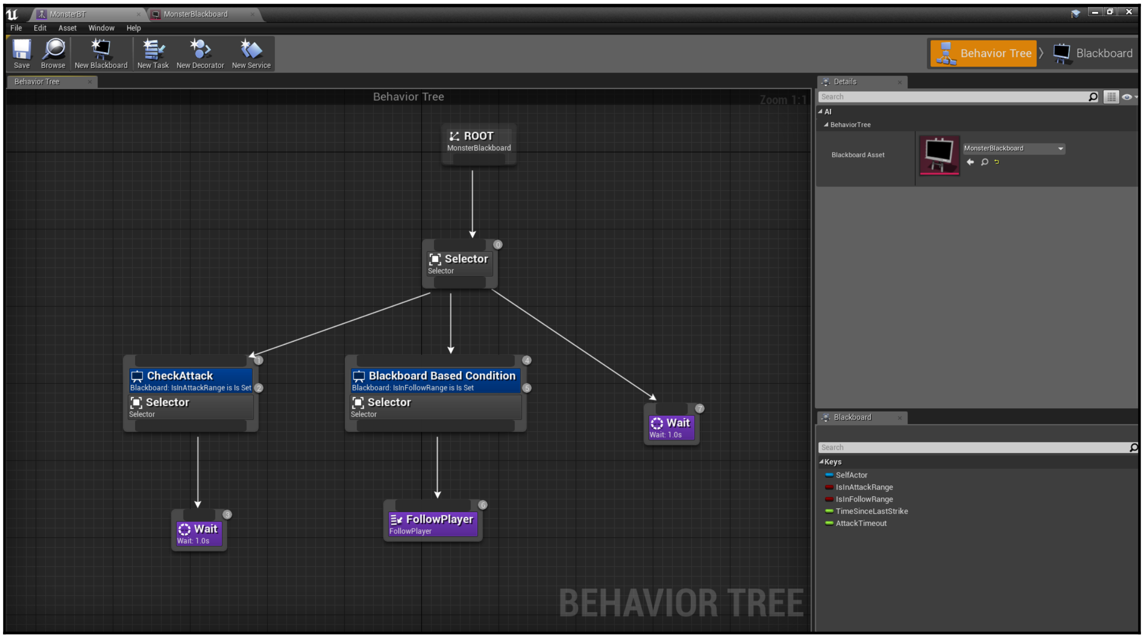
To check which state it needs to be in, you'll need to check the **Blackboard** value, which you do by using a decorator. To do this, click the bottom of the **Selector** and drag off a new node to the left like you did when you created that node, and choose a **Composite Selector** node this time. This node allows you to right-click; choose **Add Decorator...**, and make sure you choose the **Blackboard** type. Once you add it, you can select the blue Decorator on top. You should be able to check the Key Query **IsSet** and select the value you want to check, in this case **IsInAttackRange** (if it doesn't show up, make sure **MonsterBlackboard** is set in the details as the blackboard; it should be set automatically normally):



The attack node will eventually go to an **Attack** task, but for now, I just put in a **Wait** as a placeholder (a built-in task that allows you to specify a wait time in seconds).

To the right of it, you'll also want to add another **Composite** with a **Decorator** that checks for **IsInFollowRange**. This will use the new task you created (if it doesn't show up, make sure you have compiled your code and that there aren't any errors).

To the right of that, I added a **Wait** task in the event that both cases fail. When you're done, you should have something like this:



Now you're ready to go back and modify your existing code to use all this.

Updating the MonsterAIController

You'll be adding a lot more functionality to your `AIController` class now to support the Behavior Tree:

1. Your new `MonsterAIController.h` should look like this:

```
UCLASS()
class GOLDENEGG_API AMonsterAIController : public AAIController
{
    GENERATED_BODY()
public:
    AMonsterAIController(const FObjectInitializer&
ObjectInitializer);

    virtual void Possess(class APawn* InPawn) override;
```

```
        virtual void UnPossess() override;

        UBehaviorTreeComponent* BehaviorTreeCmp;

        UBlackboardComponent* BlackboardCmp;

        //Start following the player
        void StartFollowingPlayer();
        void SetFollowRange(bool val);
        void SetAttackRange(bool val);
    };
```

Also make sure you add `#include "BehaviorTree/BehaviorTreeComponent.h"` at the top of the file. Here, you are overriding the constructor as well as the `Possess` and `UnPossess` classes. The `SetFollowRange` and `SetAttackRange` functions are new and let you set the **Blackboard** values.

2. Add the following functions to `MonsterAIController.cpp`:

```
AMonsterAIController::AMonsterAIController(const class
FObjectInitializer& ObjectInitializer)
    : Super(ObjectInitializer)
{
    BehaviorTreeCmp =
ObjectInitializer.CreateDefaultSubobject<UBehaviorTreeComponent>(th
is, TEXT("MonsterBT"));
    BlackboardCmp =
ObjectInitializer.CreateDefaultSubobject<UBlackboardComponent>(this
, TEXT("MonsterBlackboard"));
}

void AMonsterAIController::Possess(class APawn* InPawn)
{
    Super::Possess(InPawn);

    AMonster* Monster = Cast<AMonster>(InPawn);
    if (Monster)
    {
        if (Monster->BehaviorTree->BlackboardAsset)
        {
BlackboardCmp->InitializeBlackboard(*Monster->BehaviorTree->Blackbo
ardAsset);
        }

        BehaviorTreeCmp->StartTree(*Monster->BehaviorTree);
    }
```

```
}


void AMonsterAIController::UnPossess()
{
    Super::UnPossess();

    BehaviorTreeCmp->StopTree();
}

void AMonsterAIController::SetFollowRange(bool val)
{
    BlackboardCmp->SetValueAsBool("IsInFollowRange", val);
}

void AMonsterAIController::SetAttackRange(bool val)
{
    BlackboardCmp->SetValueAsBool("IsInAttackRange", val);
}
```

Also add the following lines at the top of the file:

```
#include "Monster.h"
#include "BehaviorTree/BehaviorTree.h"
#include "BehaviorTree/BlackboardComponent.h"
```

`StartFollowingPlayer` remains the same so it is not listed here, but make sure you leave that in there! Now it's time to update your `Monster` class (you won't be able to compile until you do that).

Updating the Monster class

We will be doing the following updates in the `Monster` class:

- In `Monster.h`, the only change you'll be making is to add the following lines of code:

```
UPROPERTY(EditDefaultsOnly, Category = "AI")
    class UBehaviorTree* BehaviorTree;
```

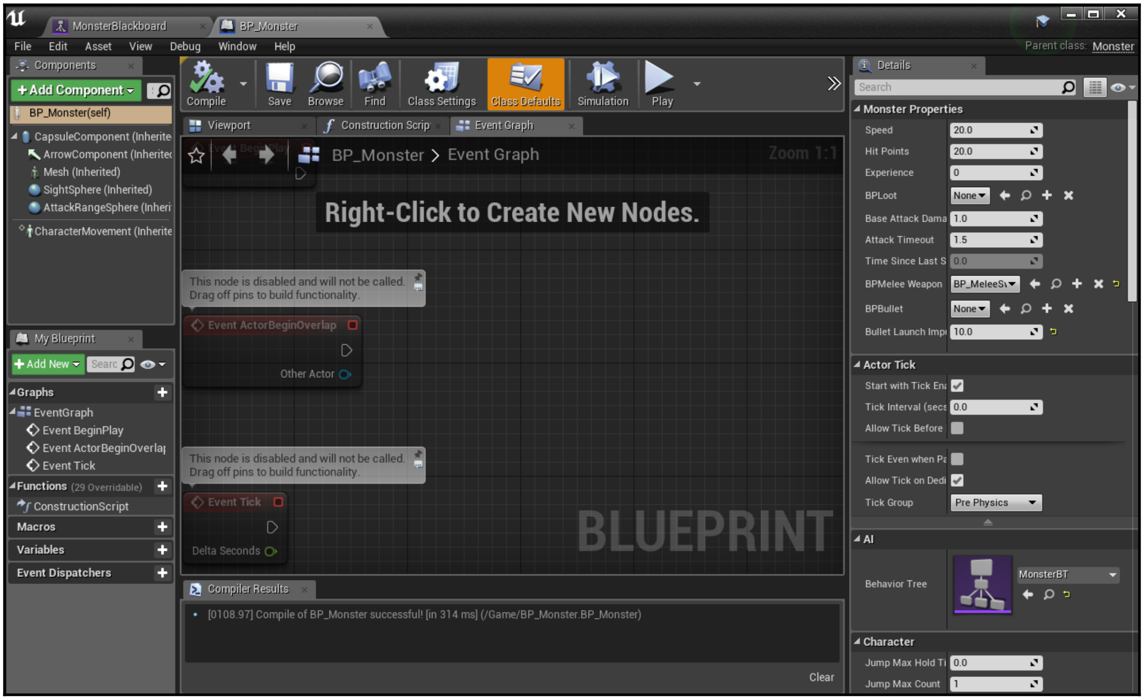
- In `Monster.cpp`, you'll be making some big changes to the `Tick()` function, so here is the full version:

```
// Called every frame
void AMonster::Tick(float DeltaSeconds)
{
    Super::Tick(DeltaSeconds);

    // move the monster towards the player
    AAvatar *avatar = Cast<AAvatar>(
        UGameplayStatics::GetPlayerPawn(GetWorld(), 0));
    if (!avatar) return;

    FVector playerPos = avatar->GetActorLocation();
    FVector toPlayer = playerPos - GetActorLocation();
    float distanceToPlayer = toPlayer.Size();
    AMonsterAIController* controller =
Cast<AMonsterAIController>(GetController());

    // If the player is not the SightSphere of the monster,
    // go back
    if (distanceToPlayer > SightSphere->GetScaledSphereRadius())
    {
        // If the player is OS, then the enemy cannot chase
        if (controller != nullptr)
        {
            controller->SetAttackRange(false);
            controller->SetFollowRange(false);
        }
        return;
    }

    toPlayer /= distanceToPlayer;  // normalizes the vector

                                   // At least face the target
                                   // Gets you the rotator to turn
something
                                   // that looks in the `toPlayer`
direction
    FRotator toPlayerRotation = toPlayer.Rotation();
    toPlayerRotation.Pitch = 0; // 0 off the pitch
    RootComponent->SetWorldRotation(toPlayerRotation);

    if (isInAttackRange(distanceToPlayer))
    {
        if (controller != nullptr)
        {
            controller->SetAttackRange(true);
```

```
            }
            // Perform the attack
            if (!TimeSinceLastStrike)
            {
                Attack(avatar);
            }

            TimeSinceLastStrike += DeltaSeconds;
            if (TimeSinceLastStrike > AttackTimeout)
            {
                TimeSinceLastStrike = 0;
            }

            return;  // nothing else to do
        }
        else
        {
            // not in attack range, so walk towards player
            //AddMovementInput(toPlayer, Speed*DeltaSeconds);
            if (controller != nullptr)
            {
                controller->SetAttackRange(false);
                controller->SetFollowRange(true);
            }
        }
    }
```

The changes are to set the values for both the attack and follow ranges. The code for attacking is still in there, but if you move **TimeSinceLastStrike** and **AttackTimeout** into the **Blackboard**, you can use that to move all that functionality into a `BTTask`. Now make sure everything compiles.

- Once it compiles, you need to open the `BP_Monster` **Blueprint** and set the Behavior Tree like this (it can also be set on individual Monsters if you want them to be different):



Also make sure the **AI Controller** is set to **MonsterAIController**. If you run the game at this point, the functionality should be the same, but the Behavior Tree will be controlling the following of the player.

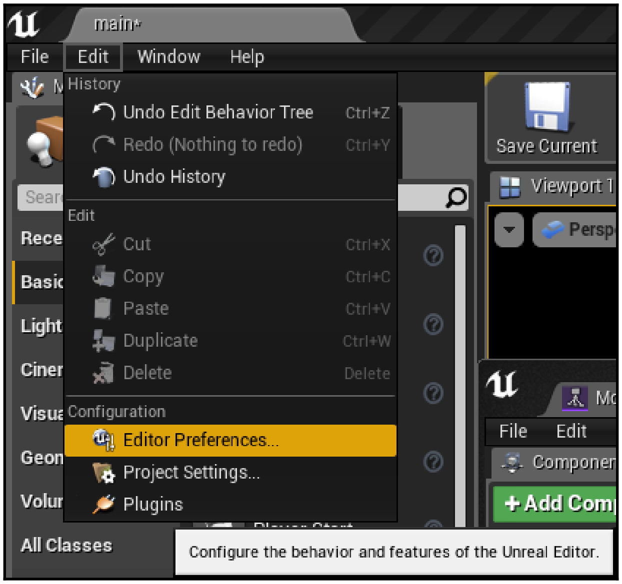
If you'd like to learn more, look into moving the `Attack` code into a `BTTask` class, and also look into what the monsters can do while you're out of range (read the next section for something that might help with that).

Environment Query Systems

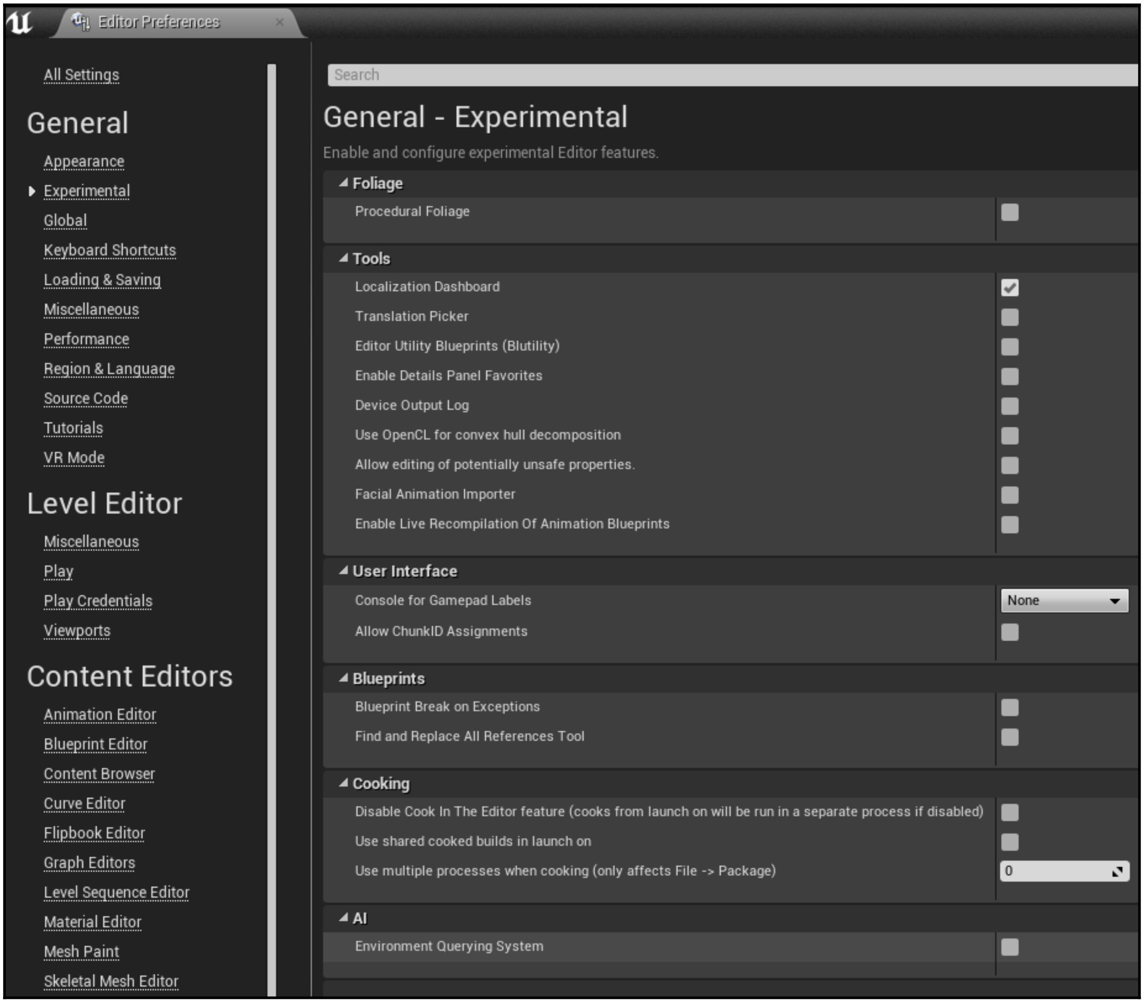
Environment Query Systems (**EQS**) are new and still experimental. They allow you to create a query in your Behavior Tree to search through the items in a level and find one that best fits the criteria you set up. Maybe you want your monsters to wander between set waypoints you set up instead of standing still when the player is out of range. You can set up a query to look for the closest one, or use some other criteria. EQS allow you to do this.

You need to enable this in the **Settings** to be able to use them. To do this, perform the following steps:

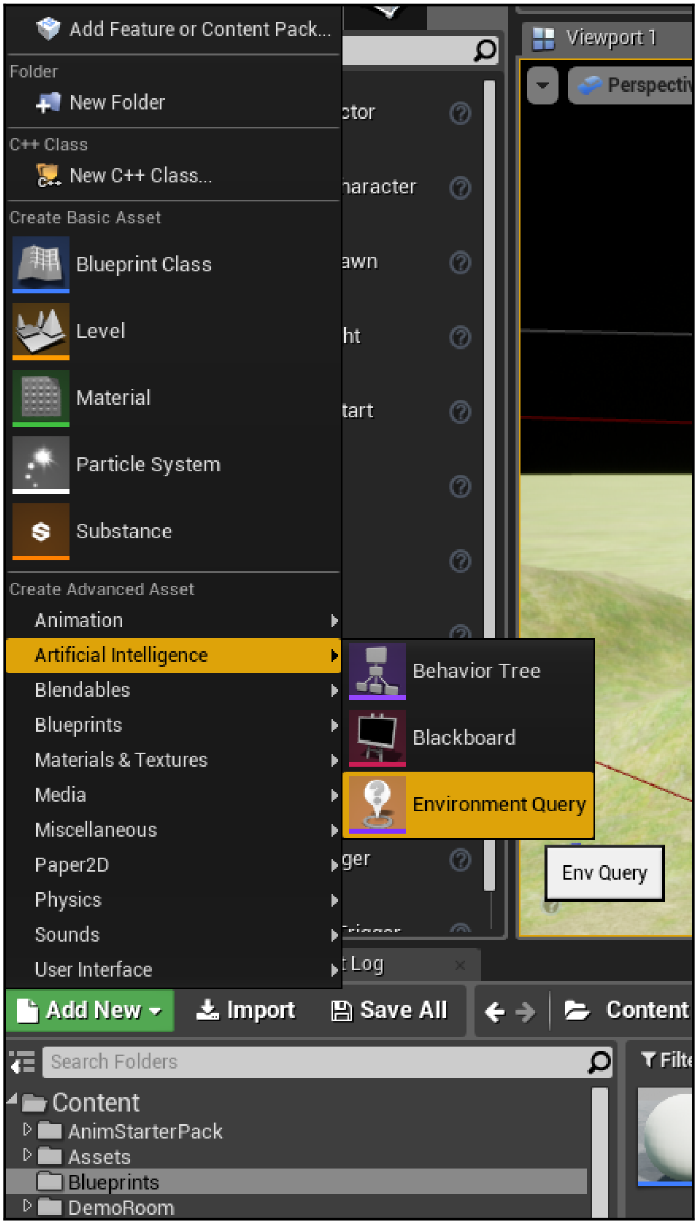
1. Go to **Edit** | **Editor Preferences**:



2. Under **Experimental | AI**, check off **Environment Querying System**:



3. Add a new query by going to **Add New** | **Artificial Intelligence**. **Environment Query** will now appear under **Behavior Tree** and **Blackboard**:



You also need to create a `Context` and a `Generator` in **Blueprints** (the `Generator` will get all the items of a specific type, such as waypoints). To actually run the query, you create a **Run EQS Query** task node in your Behavior Tree. For more information on how Environment Query Systems work, see the Unreal documentation at `https://docs.unrealengine.com/en-us/Engine/AI/EnvironmentQuerySystem`.

Flocking

If you have a lot of monsters on the screen all moving at once, you'll want them to move in a way that looks realistic. You don't want them walking into each other or all going off in different directions.

AI researchers have looked into this and come up with algorithms to handle this realistically. They're called flocking algorithms because they're based on the behavior of a flock of birds.

When moving together, monsters have to think of more than just getting to the same goal. They also have to take into account the monsters they are moving with. They have to make sure they don't get too close to the monsters immediately surrounding them, nor should they move too far away or they'll drift apart.

In many cases, there is one monster selected as the leader. That monster heads toward the goal, and the others focus on following that leader.

There are many good references on flocking online. It's not built into UE4, but you can buy extensions or program your own flocking system.

Introduction to machine learning and neural networks

Machine learning and neural networks are huge topics, so I'll only be giving a brief introduction here. Machine Learning is how you can teach a program to figure out how to respond to something, instead of just giving it rules. There are many different algorithms for doing this, but they all require a lot of sample data.

Basically, you give the learning program a large amount of example cases (the more the better), *and* the best results for each case. You can rate them in different ways. By looking at so many cases, it can make the best guess on similar cases based on results it has seen in the past. With enough training data, the results can be very good, although you can still run into cases it won't work well for.

Since this requires so much data (not to mention processing power), except in rare cases, this is done by game companies before the game is shipped (if it's done at all—this sort of thing tends to get cut in favor of deadlines). The training is done offline, and the program has already learned what to do.

Neural Networks are a specific type of machine learning made to emulate the way the brain processes data. There are nodes that work like neurons. There can be multiple layers of nodes and each layer processes the results of the previous one.

Data is sent across multiple nodes, and each node adjusts that data based on some threshold amount. Only data can be passed back (or forward) to the nodes, which then adjust those threshold values to get more accurate results to the training data. Once they've been trained, those threshold values can be used for future decisions.

While we're still very far from making a true AI, neural networks have been used with interesting results. Neural networks have been trained on music of a specific genre and have then generated very impressive (and original) music that sounds similar to the genre it was trained on. I've also heard of neural networks being written to attempt to write books. I think we're still a long way from a neural network that can write UE4 programs, though!

Genetic algorithms

Recall your high school biology; you probably learned about genetics. Chromosomes from two different parents combine to create a child that combines the DNA from both parents, and random genetic mutations can also make changes. Genetic algorithms are based on the same principles.

Like in Darwin's survival of the fittest, you can do something very similar in code. Genetic algorithms have three basic principles:

- **Selection**: You pick the examples that have the best results, and those are the basis for the next generation.
- **Crossover**: Two of these selected examples are then combined to create a child that is a product of both, just like in biology.
- **Random genetic mutations are introduced**: There could be some good traits that the old ones didn't have, or that got thrown out because those traits were overwhelmed by other traits. This means you don't miss out on some potentially great traits just because they weren't in the original population.

Summary

As you've seen, AI is a huge topic, and we've only touched on the basics here. We've gone over the basics of pathfinding (with the NavMesh), Behavior Trees, environment query systems, Flocking, machine learning and neural networks, and Genetic algorithms. There are entire books out there if you want to learn more, as well as numerous websites, such as `http://aigamedev.com/`, and articles on `https://www.gamasutra.com`.

In the next section, we will learn to cast spells to defend your player from the monsters.

13
Spell Book

The player does not yet have a means to defend himself. We will now equip the player with a very useful and interesting way of doing so, called magic spells. Magic spells will be used by the player to affect monsters nearby, so you can now damage them.

We'll begin the chapter by describing how to create our own particle systems. We'll then move on to wrap the particle emitter into a `Spell` class, and write a `CastSpell` function for the avatar to be able to actually `CastSpells`.

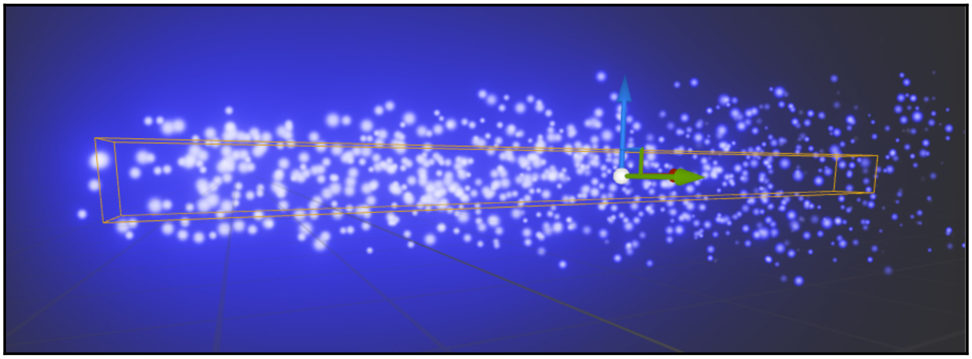
The following topics will be covered in this chapter:

- What is a spell?
- Particle systems
- Spell class actor
- Attaching right mouse click to CastSpell
- Creating other spells

What is a spell?

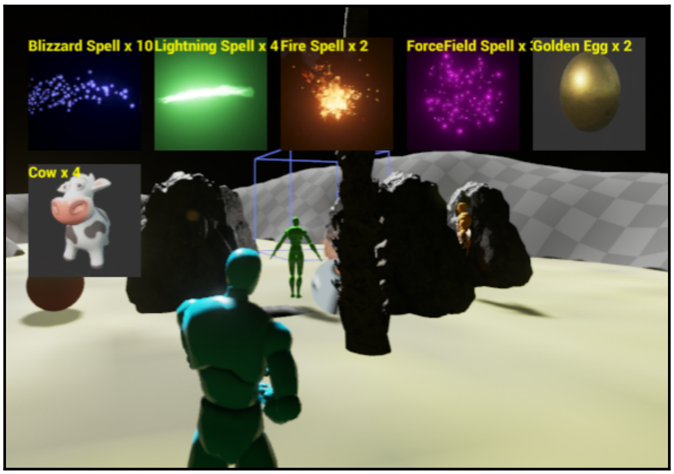
Practically, spells will be a combination of a particle system with an area of effect represented by a bounding volume. The bounding volume is checked for actors contained in each frame. When an actor is within the bounding volume of a spell, then that actor is affected by that spell.

The following is a screenshot of the blizzard spell, with the bounding volume highlighted in orange:



The blizzard spell has a long, box-shaped bounding volume. In each frame, the bounding volume is checked for contained actors. Any actor contained in the spell's bounding volume is going to be affected by that spell for that frame only. If the actor moves outside the spell's bounding volume, the actor will no longer be affected by that spell. Remember, the spell's particle system is a visualization only; the particles themselves are not what will affect game actors.

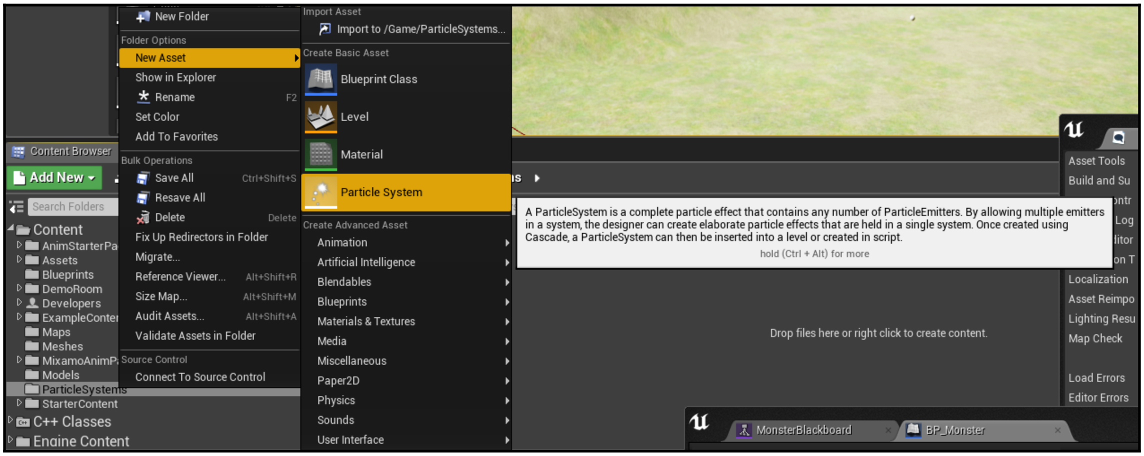
The `PickupItem` class we created in `Chapter 8`, *Actors and Pawns*, can be used to allow the player to pick up items representing the spells. We will extend the `PickupItem` class and attach the blueprint of a spell to cast each `PickupItem`. Clicking on a spell's widget from the HUD will cast it. The interface will look something like this:



Setting up particle systems

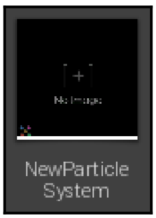
First, we need a place to put all our snazzy effects. To do so, we will follow these steps:

1. In your **Content Browser** tab, right-click on the **Content** root and create a new folder called `ParticleSystems`.
2. Right-click on that new folder, and select **New Asset** | **Particle System**, as shown in the following screenshot:

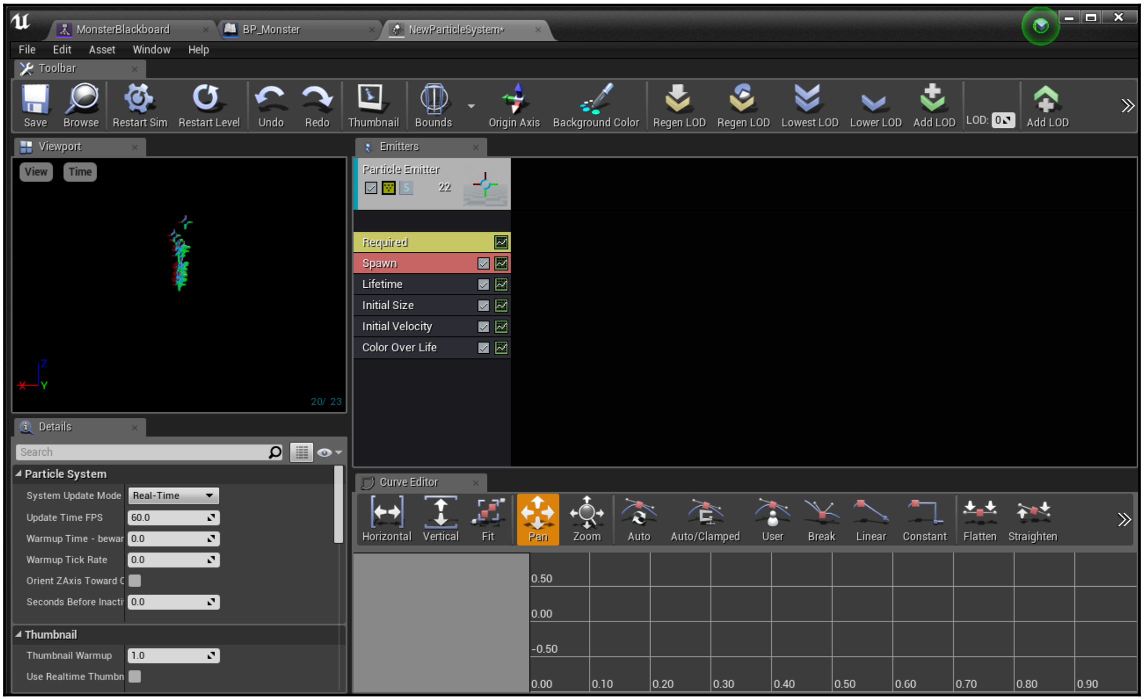


See this Unreal Engine 4 particle systems guide for information on how Unreal particle emitters work:
`https://www.youtube.com/watch?v=OXK2Xbd7D9w&index=1&list=PLZlv_N0_O1gYDLyB3LVfjYIcbBe8NqR8t`.

3. Double-click on the **NewParticleSystem** icon that appears, as shown in the following screenshot:



Once you are done with the preceding steps, you will be in Cascade, the particle editor. The environment is shown in the following screenshot:



There are several different panes here, each of which shows different information. They are as follows:

- At the top left is the **Viewport** pane. This shows you an animation of the current emitter as it's currently working.
- At the right is the **Emitters** pane. Inside it, you can see a single object called **Particle Emitter** (you can have more than one emitter in your particle system, but we don't want that now). The listing of modules of **Particle Emitter** appears listed under it. From the preceding screenshot, we have the **Required**, **Spawn**, **Lifetime**, **Initial Size**, **Initial Velocity**, and **Color Over Life** modules.

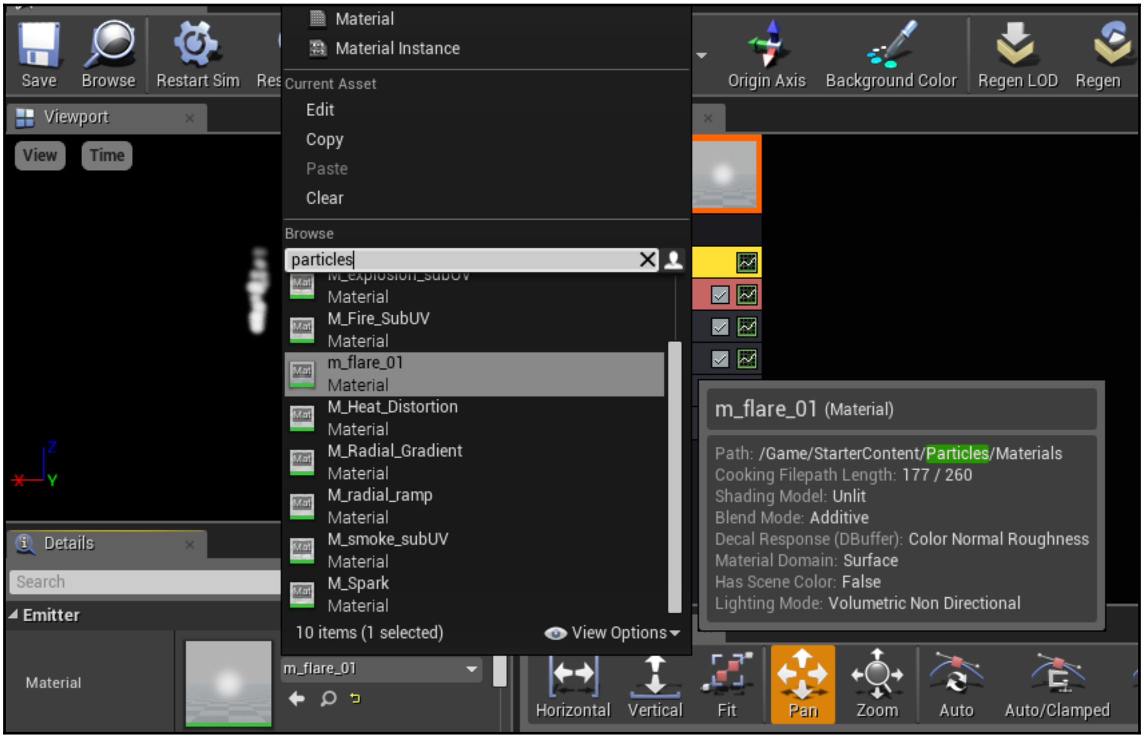
Changing particle properties

The default particle emitter emits crosshair-like shapes. We want to change that to something more interesting. To do this, follow these steps:

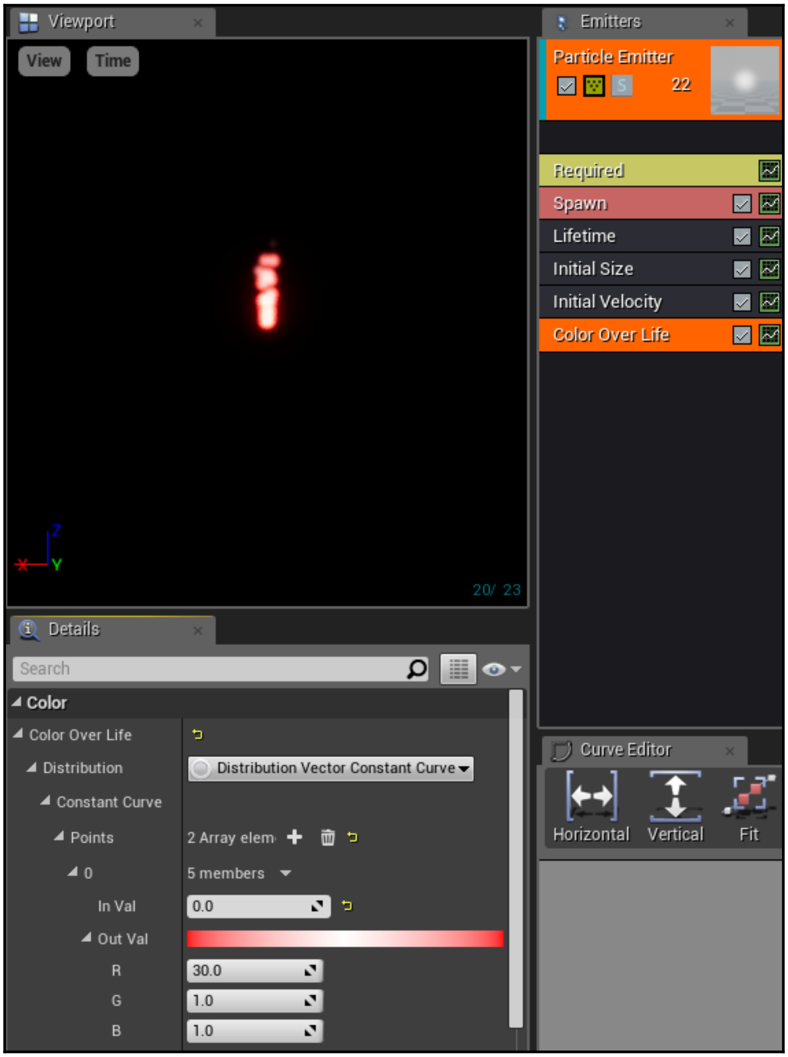
1. Click on the yellow **Required** box under the **Emitters** panel, then open the **Material** drop-down in the **Details** panel.

   A list of all the available particle materials will pop up (you can type `particles` in the top to make it easier to find the ones you want).

2. Choose the **m_flare_01** option to create our first particle system, as shown in the following screenshot:

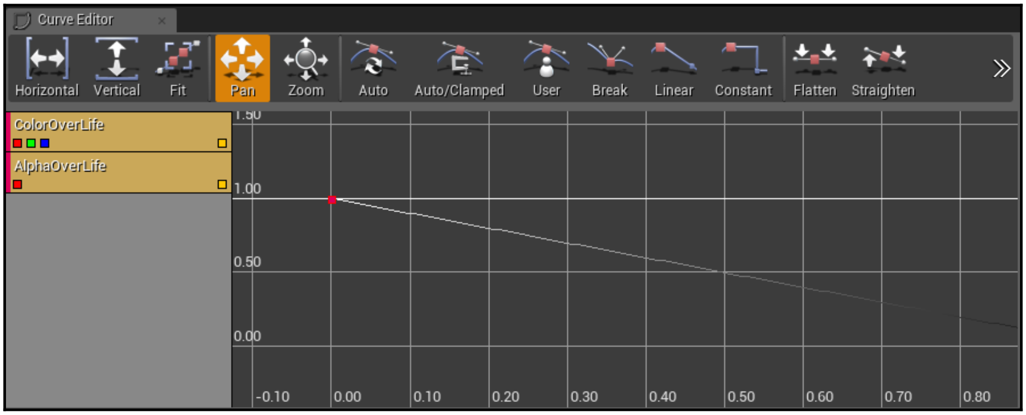


3. Now, let's change the behavior of the particle system. Click on the **Color Over Life** entry under the **Emitters** pane. The **Details** pane at the bottom shows information about the different parameters, as shown in the following screenshot:

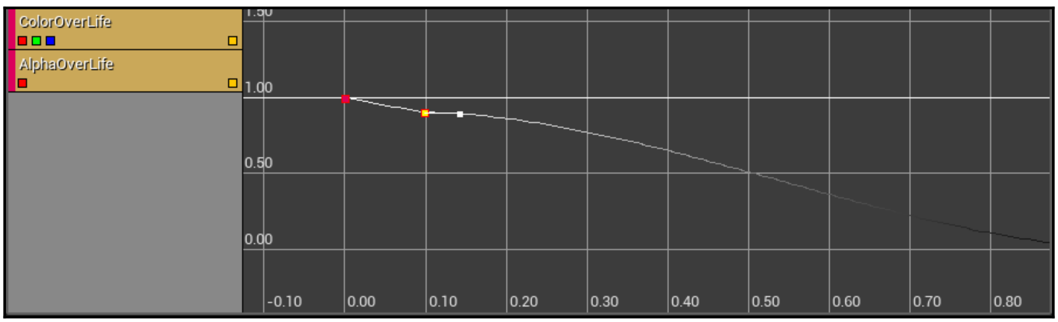


4. In the **Details** pane of **Color Over Life** entry, I increased **R**, but not **G** and not **B**. This gives the particle system a reddish glow. (**R** is red, **G** is green, and **B** is blue). You can see the color on the bar.

Instead of editing the raw numbers, however, you can actually change the particle color more visually. If you click on the greenish zigzag button beside the **Color Over Life** entry under **Emitters**, you will see the graph for **Color Over Life** displayed in the **Curve Editor** tab, as shown in the following screenshot:



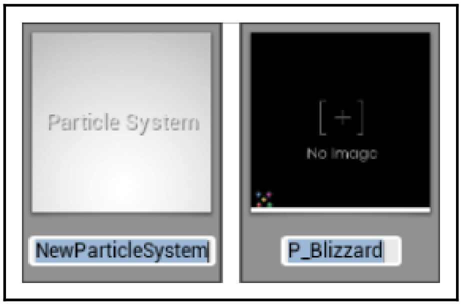
We can now change the **Color Over Life** parameters. The graph in the **Curve Editor** tab displays the emitted color versus the amount of time the particle has been alive. You can adjust the values by dragging the points around. Pressing *Ctrl* + left mouse button adds a new point to a line (if it doesn't work, click in the yellow box to deselect **AlphaOverLife** and make sure only **ColorOverLife** is selected):



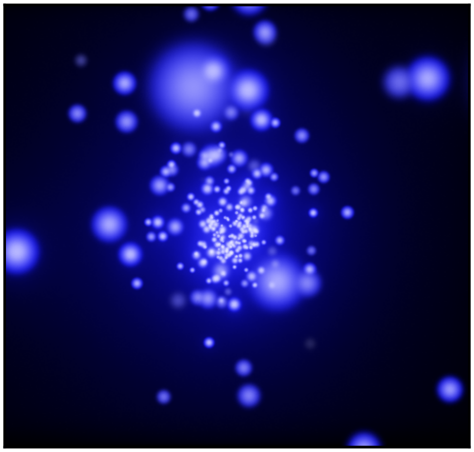
You can play around with the particle emitter settings to create your own spell visualizations.

Settings for the blizzard spell

At this point, we should rename our particle system from **NewParticleSystem** to something more descriptive. Let's rename it `P_Blizzard`.

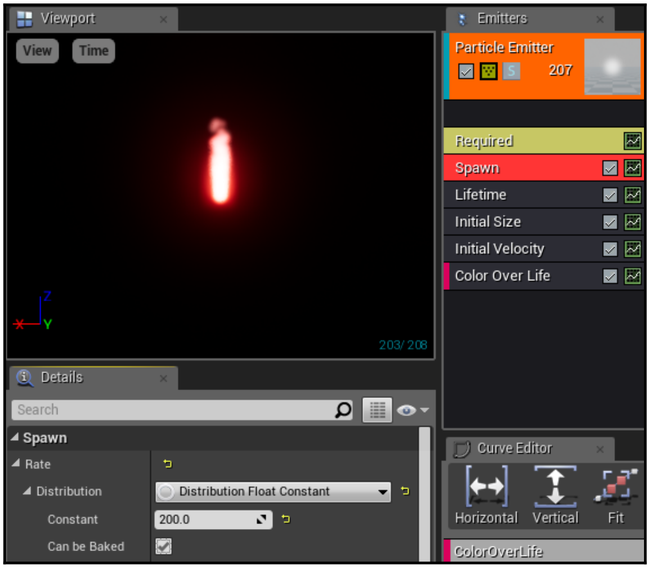


You can rename your particle system by simply clicking on it and pressing *F2, as shown below*:

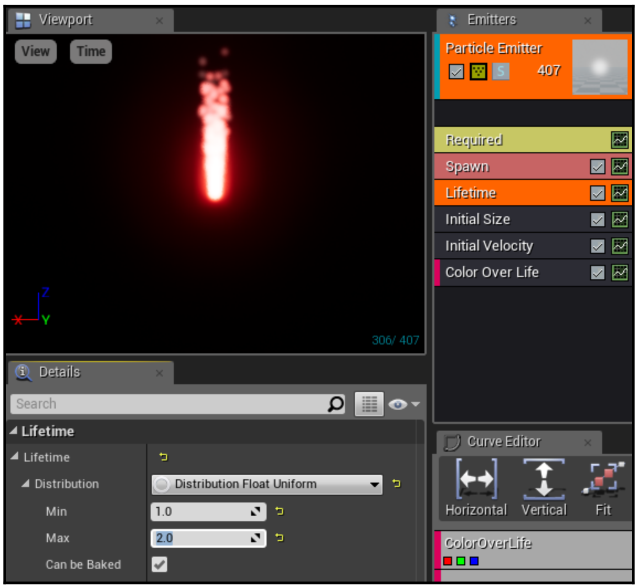


We will tweak some of the settings to get a blizzard particle effect spell. Perform the following steps:

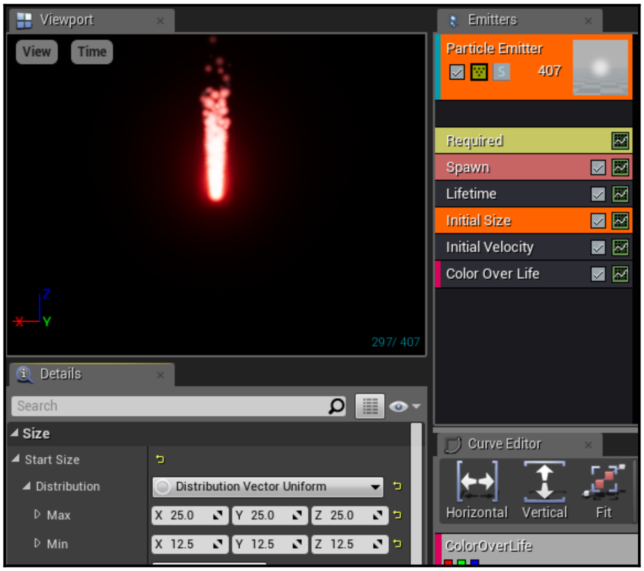
1. Go back into the P_Blizzard particle system to edit it.
2. Under the **Spawn** module, change the spawn rate to `200.0`. This increases the density of the visualization, as shown here:



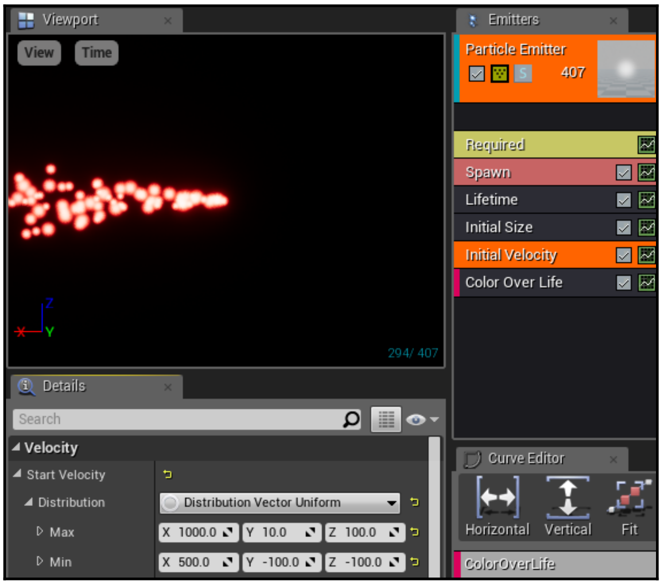
3. Under the **Lifetime** module, increase the **Max** property from `1.0` to `2.0`, as shown in the following screenshot. This introduces some variation to the length of time a particle will live, with some of the emitted particles living longer than others:



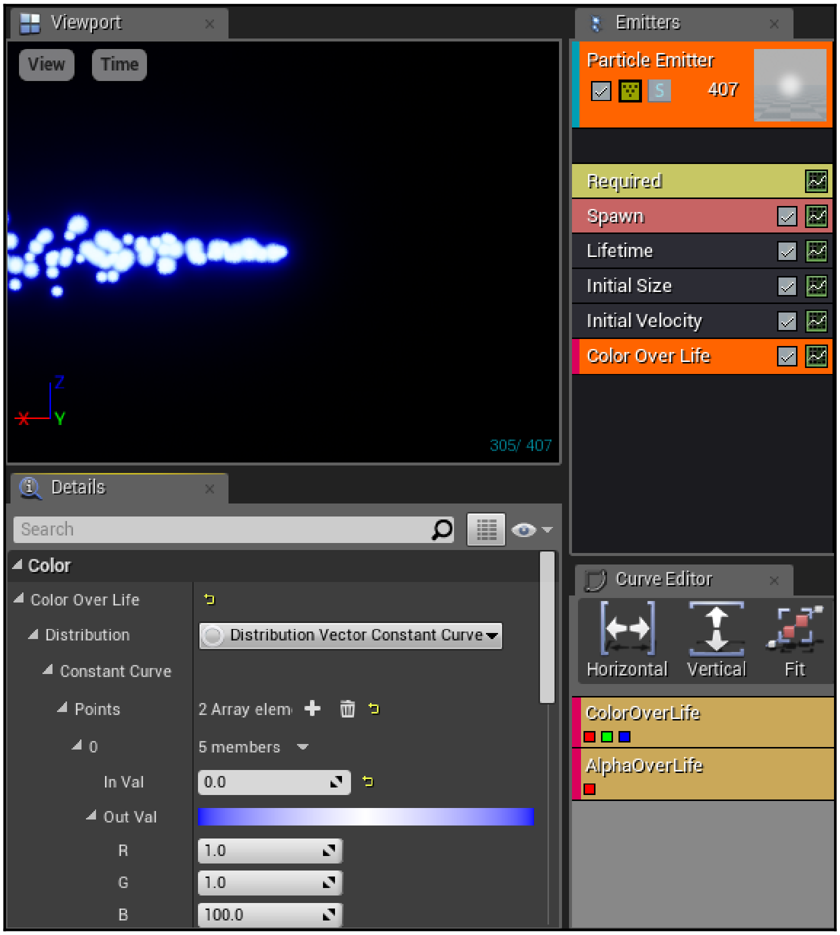
4. Under the **Initial Size** module, change the **Min** property size to `12.5` in **X**, **Y**, and **Z**, as shown in the following screenshot:



5. Under the **Initial Velocity** module, change the **Min** / **Max** values to the values shown here:

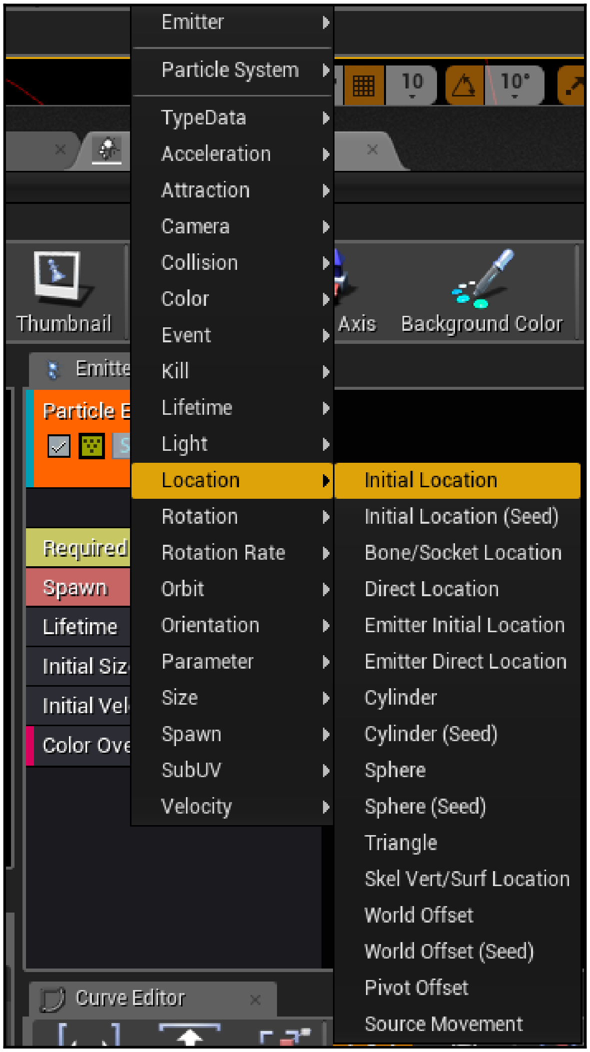


6. The reason we're having the blizzard blow in **+X** is because the player's forward direction starts out in **+X**. Since the spell will come from the player's hands, we want the spell to point in the same direction as the player.
7. Under the **Color Over Life** menu, change the blue (**B**) value to `100.0`. Also change R back to `1.0`. You will see an instant change to a blue glow:

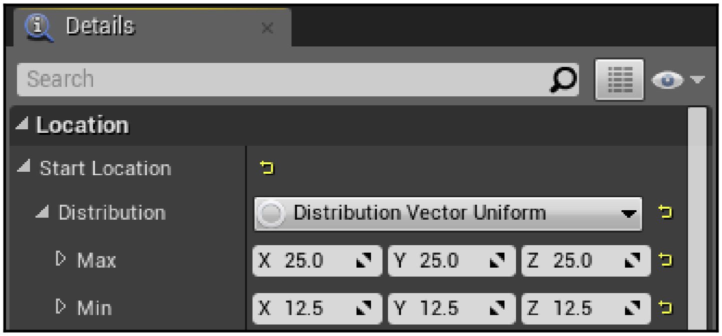


Now it's starting to look magical!

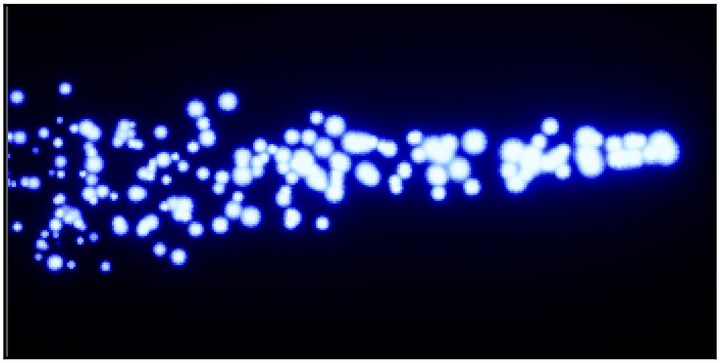
8. Right-click on the blackish area below the **Color Over Life** module. Choose **Location** | **Initial Location**, shown in the screenshot:



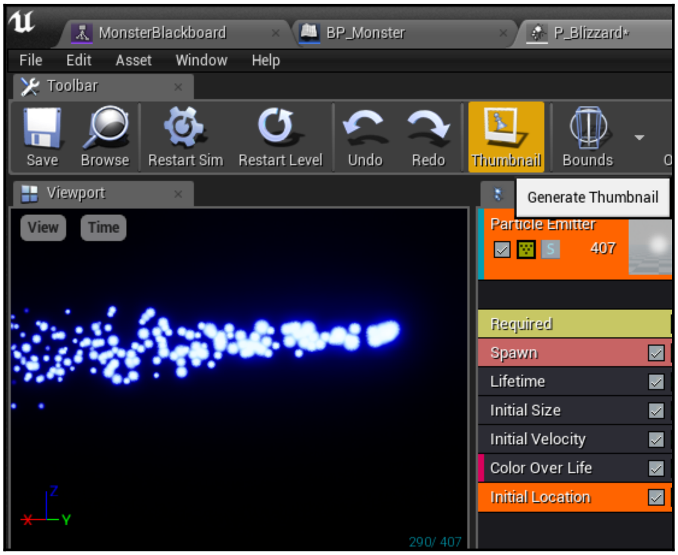
9. Enter values under **Start Location** | **Distribution** as shown in the following screenshot:



10. You should have a blizzard that looks like this:



11. Move the camera to a position you like, and then click on the **Thumbnail** option in the top menu bar. This will generate a **Thumbnail** icon for your particle system in the **Content Browser** tab, as shown in the following screenshot:



Spell class actor

The `Spell` class will ultimately do damage to all monsters. Toward that end, we need to contain both a particle system and a bounding box inside the `Spell` class actor. When a `Spell` class is cast by the avatar, the `Spell` object will be instantiated into the level and start `Tick()` functioning. On every `Tick()` of the `Spell` object, any monster contained inside the spell's bounding volume will be affected by that `Spell`.

The `Spell` class should look something like the following code:

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Components/BoxComponent.h"
#include "Runtime/Engine/Classes/Particles/ParticleSystemComponent.h"
#include "Spell.generated.h"

UCLASS()
class GOLDENEGG_API ASpell : public AActor
{
    GENERATED_BODY()
```

```
public:
    ASpell(const FObjectInitializer&amp; ObjectInitializer);
    // box defining volume of damage
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
        Spell)
        UBoxComponent* ProxBox;

    // the particle visualization of the spell
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
        Spell)
        UParticleSystemComponent* Particles;

    // How much damage the spell does per second
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Spell)
        float DamagePerSecond;

    // How long the spell lasts
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Spell)
        float Duration;

    // Length of time the spell has been alive in the level
    float TimeAlive;

    // The original caster of the spell (so player doesn't
    // hit self)
    AActor* Caster;

    // Parents this spell to a caster actor
    void SetCaster(AActor* caster);

    // Runs each frame. override the Tick function to deal damage
    // to anything in ProxBox each frame.
    virtual void Tick(float DeltaSeconds) override;
};
```

There are only three functions we need to worry about implementing, namely the `ASpell::ASpell()` constructor, the `ASpell::SetCaster()` function, and the `ASpell::Tick()` function.

Open the `Spell.cpp` file. Underneath the include line for `Spell.h`, add a line to include the `Monster.h` file, so we can access the definition of `Monster` objects inside the `Spell.cpp` file (as well as a couple of other includes), as shown in the following line of code:

```
#include "Monster.h"
#include "Kismet/GameplayStatics.h"
#include "Components/CapsuleComponent.h"
```

First, the constructor, which sets up the spell and initializes all components, is shown in the following code:

```
ASpell::ASpell(const FObjectInitializer&amp; ObjectInitializer)
 : Super(ObjectInitializer)
{
 ProxBox = ObjectInitializer.CreateDefaultSubobject<UBoxComponent>(this,
 TEXT("ProxBox"));
 Particles =
ObjectInitializer.CreateDefaultSubobject<UParticleSystemComponent>(this,
 TEXT("ParticleSystem"));

 // The Particles are the root component, and the ProxBox
 // is a child of the Particle system.
 // If it were the other way around, scaling the ProxBox
 // would also scale the Particles, which we don't want
 RootComponent = Particles;
 ProxBox->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepWorldTransform);

 Duration = 3;
 DamagePerSecond = 1;
 TimeAlive = 0;

 PrimaryActorTick.bCanEverTick = true;//required for spells to
 // tick!
}
```

Of particular importance is the last line here, `PrimaryActorTick.bCanEverTick = true`. If you don't set that, your `Spell` objects won't ever have `Tick()` called.

Next, we have the `SetCaster()` method. This is called so that the person who casts the spell is known to the `Spell` object. We can ensure that the caster can't hurt himself with his own spells by using the following code:

```
void ASpell::SetCaster(AActor *caster)
{
 Caster = caster;
 RootComponent->AttachToComponent(caster->GetRootComponent(),
FAttachmentTransformRules::KeepRelativeTransform);
}
```

Finally, we have the `ASpell::Tick()` method, which actually deals damage to all contained actors, as shown in the following code:

```
void ASpell::Tick(float DeltaSeconds)
{
    Super::Tick(DeltaSeconds);

    // search the proxbox for all actors in the volume.
    TArray<AActor*> actors;
    ProxBox->GetOverlappingActors(actors);

    // damage each actor the box overlaps
    for (int c = 0; c < actors.Num(); c++)
    {
        // don't damage the spell caster
        if (actors[c] != Caster)
        {
            // Only apply the damage if the box is overlapping
            // the actors ROOT component.
            // This way damage doesn't get applied for simply
            // overlapping the SightSphere of a monster
            AMonster *monster = Cast<AMonster>(actors[c]);

            if (monster &amp;&amp;
ProxBox->IsOverlappingComponent(Cast<UPrimitiveComponent>(monster->GetCapsu
leComponent())))
            {
                monster->TakeDamage(DamagePerSecond*DeltaSeconds,
                    FDamageEvent(), 0, this);
            }

            // to damage other class types, try a checked cast
            // here..
        }
    }

    TimeAlive += DeltaSeconds;
    if (TimeAlive > Duration)
    {
        Destroy();
    }
}
```

The `ASpell::Tick()` function does a number of things, as follows:

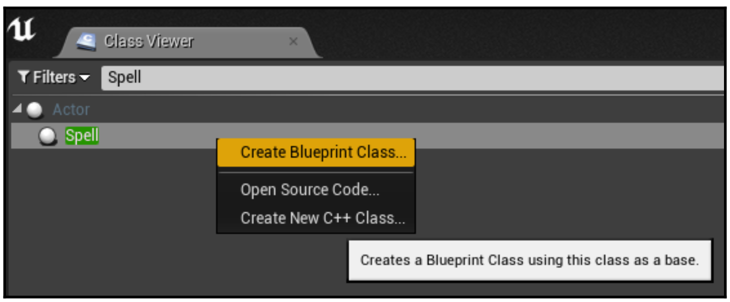
- It gets all actors overlapping `ProxBox`. Any actor that is not the caster gets damaged if the component overlapped is the root component of that object. The reason we have to check for overlapping with the root component is because, if we don't, the spell might overlap the monster's `SightSphere`, which means we will get hits from very far away, which we don't want.
- Notice that if we had another class of thing that should get damaged, we would have to attempt a cast to each object type specifically. Each class type might have a different type of bounding volume that should be collided with; other types might not even have `CapsuleComponent` (they might have `ProxBox` or `ProxSphere`).
- It increases the amount of time the spell has been alive for. If the spell exceeds the duration it is allotted to be cast for, it is removed from the level.

Now, let's focus on how the player can acquire spells, by creating an individual `PickupItem` for each spell object that the player can pick up.

Blueprinting our spells

Compile and run your C++ project with the `Spell` class that we just added. We need to create blueprints for each of the spells we want to be able to cast. To do this, follow these steps:

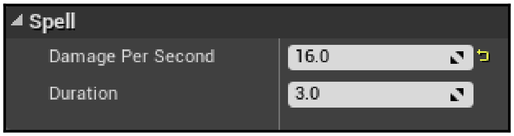
1. In the **Class Viewer** tab, start to type `Spell`, and you should see your **Spell** class appear
2. Right-click on **Spell**, and create a blueprint called **BP_Spell_Blizzard** as shown in the following screenshot:



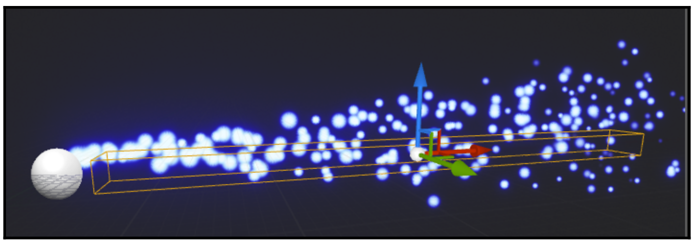
3. If it doesn't open automatically, double-click to open it.
4. Inside the spell's properties, choose the **P_Blizzard** spell for the particle emitter, as shown in the following screenshot:

If you can't find it, try selecting **Particles (Inherited)** under **Components**.

With **BP_SpellBlizzard(self)** selected, scroll down until you reach the **Spell** category, and update the **Damage Per Second** and **Duration** parameters to values you like, as shown in the following screenshot. Here, the blizzard spell will last `3.0` seconds, and do `16.0` damage per second. After three seconds, the blizzard will disappear:



After you have configured the **Default** properties, switch over to the **Components** tab to make some further modifications. Click on and change the shape of `ProxBox` so that its shape makes sense. The box should wrap the most intense part of the particle system, but don't get carried away in expanding its size. The `ProxBox` object shouldn't be too big, because then your blizzard spell would affect things that aren't even being touched by the blizzard. As shown in the following screenshot, a couple of outliers are OK:



Your blizzard spell is now blueprinted and ready to be used by the player.

Picking up spells

Recall that we previously programmed our inventory to display the number of pickup items the player has when the user presses *I*. We want to do more than that, however:

Items displayed when the user presses I

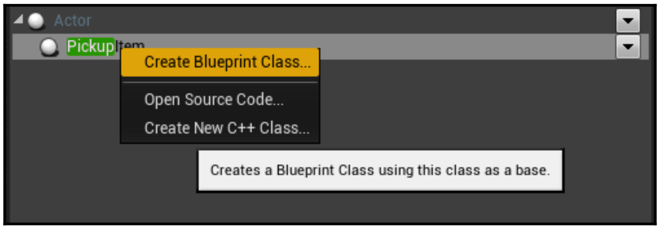
To allow the player to pick up spells, we'll modify the `PickupItem` class to include a slot for a blueprint of the spell the player casts by using the following code:

```
// inside class APickupItem:
// If this item casts a spell when used, set it here
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
UClass* Spell;
```

Once you've added the `UClass* Spell` property to the `APickupItem` class, recompile and rerun your C++ project. Now, you can proceed to make blueprints of `PickupItem` instances for your `Spell` objects.

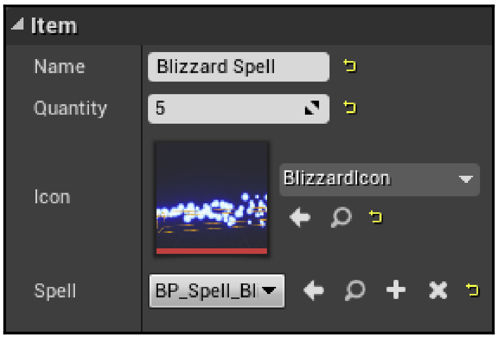
Creating blueprints for PickupItems that Cast Spells

Create a **PickupItem** blueprint called **BP_Pickup_Spell_Blizzard**, as shown in the following screenshot:

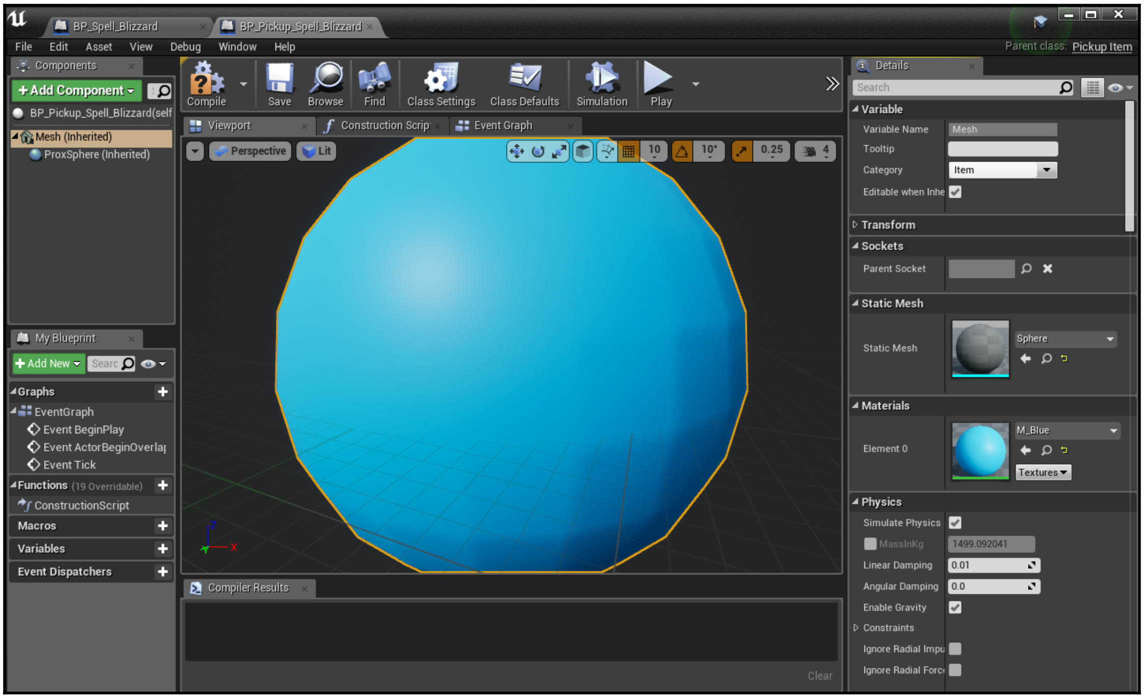


It should open automatically so you can edit its properties. I set the blizzard item's pickup properties as follows:

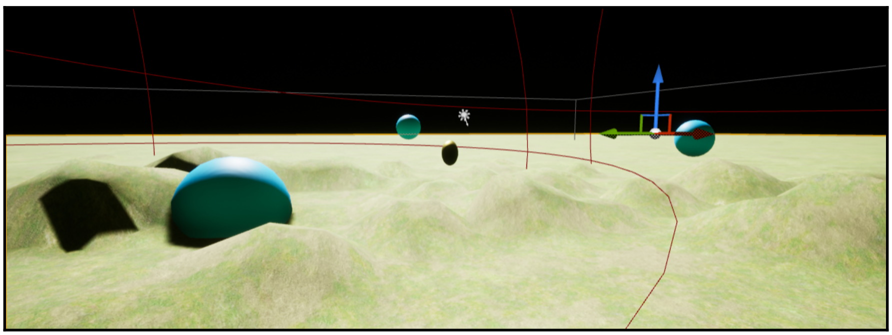
The name of the item is **Blizzard Spell**, and `5` are in each package. I took a screenshot of the blizzard particle system and imported it into the project, so the **Icon** is selected as that image. Under **Spell**, I selected **BP_Spell_Blizzard** as the name of the spell to be cast (not **BP_Pickup_Spell_Blizzard**), as shown in the following screenshot:



I selected a blue sphere for the `Mesh` class of the `PickupItem` class (you can also get an interesting one by using the **M_Water_Lake** material). For **Icon**, I took a screenshot of the blizzard spell in the particle viewer preview, saved it to disk, and imported that image to the project, as shown in the following screenshot (see the `images` folder in the **Content Browser** tab of the sample project):



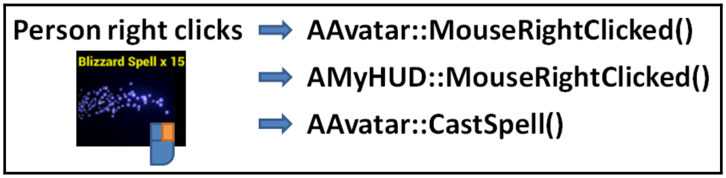
Place a few of these `PickupItem` in your level. If we pick them up, we will have some blizzard spells in our inventory (if you can't pick them up make sure you made the ProxSphere big enough):



Now, we need to activate the blizzard. Since we already attached the left mouse click in `Chapter 10`, *Inventory System and Pickup Items*, to drag the icons around, let's attach the right mouse click to casting the spell.

Attaching right mouse click to CastSpell

The right mouse click will have to go through quite a few function calls before calling the avatar's `CastSpell` method. The call graph would look something like the following screenshot:



A few things happen between right-click and spell cast. They are as follows:

- As we saw before, all user mouse and keyboard interactions are routed through the `Avatar` object. When the `Avatar` object detects a right-click, it will pass the click event to `HUD` through `AAvatar::MouseRightClicked()`.
- In `Chapter 10`, *Inventory System and Pickup Items* we used a `struct Widget` class to keep track of the items the player had picked up. `struct Widget` only had three members:

```
struct Widget
{
  Icon icon;
  FVector2D pos, size;
  ///.. and some member functions
};
```

- We will now need to add an extra property for the `struct Widget` class to remember the spell it casts.
- The `HUD` will determine if the click event was inside `Widget` in `AMyHUD::MouseRightClicked()`.
- If the click was on the `Widget` that casts a spell, the `HUD` then calls the avatar back with the request to cast that spell, by calling `AAvatar::CastSpell()`.

Writing the avatar's CastSpell function

We will implement the preceding call graph in reverse. We will start by writing the function that actually casts spells in the game, `AAvatar::CastSpell()`, as shown in the following code:

```
void AAvatar::CastSpell( UClass* bpSpell )
{
  // instantiate the spell and attach to character
  ASpell *spell = GetWorld()->SpawnActor<ASpell>(bpSpell,
   FVector(0), FRotator(0) );

  if( spell )
  {
    spell->SetCaster( this );
  }
  else
  {
    GEngine->AddOnScreenDebugMessage( 1, 5.f, FColor::Yellow,
    FString("can't cast ") + bpSpell->GetName() ); }
}
```

Also make sure to add the function to `Avatar.h` and add `#include "Spell.h"` to the top of this file.

You might find that actually calling a spell is remarkably simple. There are two basic steps to casting the spell:

1. Instantiate the spell object using the world object's `SpawnActor` function
2. Attach it to the avatar

Once the `Spell` object is instantiated, its `Tick()` function will run each frame when that spell is in the level. On each `Tick()`, the `Spell` object will automatically feel out monsters in the level and damage them. A lot happens with each line of code mentioned previously, so let's discuss each line separately.

Instantiating the spell – GetWorld()->SpawnActor()

To create the `Spell` object from the blueprint, we need to call the `SpawnActor()` function from the `World` object. The `SpawnActor()` function can take any blueprint and instantiate it within the level. Fortunately, the `Avatar` object (and indeed any `Actor` object) can get a handle to the `World` object at any time by simply calling the `GetWorld()` member function.

The line of code that brings the `Spell` object into the level is as follows:

```
ASpell *spell = GetWorld()->SpawnActor<ASpell>( bpSpell,
   FVector(0), FRotator(0) );
```

There are a couple of things to note about the preceding line of code:

- `bpSpell` must be the blueprint of a `Spell` object to create. The `<ASpell>` object in angle brackets indicates that expectation.
- The new `Spell` object starts out at the origin (`0`, `0`, `0`), and with no additional rotation applied to it. This is because we will attach the `Spell` object to the `Avatar` object, which will supply translation and direction components for the `Spell` object.

if(spell)

We always test if the call to `SpawnActor<ASpell>()` succeeds by checking `if( spell )`. If the blueprint passed to the `CastSpell` object is not actually a blueprint based on the `ASpell` class, then the `SpawnActor()` function returns a `NULL` pointer instead of a `Spell` object. If that happens, we print an error message to the screen indicating that something went wrong during spell casting.

spell->SetCaster(this)

When instantiating, if the spell does succeed, we attach the spell to the `Avatar` object by calling `spell->SetCaster( this )`. Remember, in the context of programming within the `Avatar` class, the `this` method is a reference to the `Avatar` object.

Now, how do we actually connect spell casting from UI inputs, to call the `AAvatar::CastSpell()` function in the first place? We need to do some `HUD` programming again.

Writing AMyHUD::MouseRightClicked()

The spell cast commands will ultimately come from the HUD. We need to write a C++ function that will walk through all the HUD widgets and test to see if a click is on any one of them. If the click is on a `widget` object, then that `widget` object should respond by casting its spell, if it has one assigned.

We have to extend our `Widget` object to have a variable to hold the blueprint of the spell to cast. Add a member to your `struct Widget` object by using the following code:

```
struct Widget
{
    Icon icon;
    // bpSpell is the blueprint of the spell this widget casts
    UClass *bpSpell;
    FVector2D pos, size;
    //...
};
```

Now, recall that our `PickupItem` had the blueprint of the spell it casts attached to it previously. However, when the `PickupItem` class is picked up from the level by the player, then the `PickupItem` class is destroyed, as shown in the following code:

```
// From APickupItem::Prox_Implementation():
avatar->Pickup( this ); // give this item to the avatar
// delete the pickup item from the level once it is picked up
Destroy();
```

So, we need to retain the information of what spell each `PickupItem` casts. We can do that when that `PickupItem` is first picked up.

Inside the `AAvatar` class, add an extra map to remember the blueprint of the spell that an item casts, by item name, with the following line of code:

```
// Put this in Avatar.h
TMap<FString, UClass*> Spells;
```

Now, in `AAvatar::Pickup()`, remember the class of spell the `PickupItem` class instantiates with the following line of code:

```
// the spell associated with the item
Spells.Add(item->Name, item->Spell);
```

Now, in `AAvatar::ToggleInventory()`, we can have the `Widget` object that displays on the screen. Remember what spell it is supposed to cast by looking up the `Spells` map.

Find the line where we create the widget, and modify it to add assignment of the `bpSpell` objects that the `Widget` casts, as shown in the following code:

```
// In AAvatar::ToggleInventory()
Widget w(Icon(fs, tex));
w.bpSpell = Spells[it->Key];
hud->addWidget(w);
```

Add the following function to `AMyHUD`, which we will set to run whenever the right mouse button is clicked on the icon:

```
void AMyHUD::MouseRightClicked()
{
    FVector2D mouse;
    APlayerController *PController =
GetWorld()->GetFirstPlayerController();
    PController->GetMousePosition(mouse.X, mouse.Y);
    for (int c = 0; c < widgets.Num(); c++)
    {
        if (widgets[c].hit(mouse))
        {
            AAvatar *avatar = Cast<AAvatar>(
                UGameplayStatics::GetPlayerPawn(GetWorld(), 0));
            if (widgets[c].bpSpell)
                avatar->CastSpell(widgets[c].bpSpell);
        }
    }
}
```

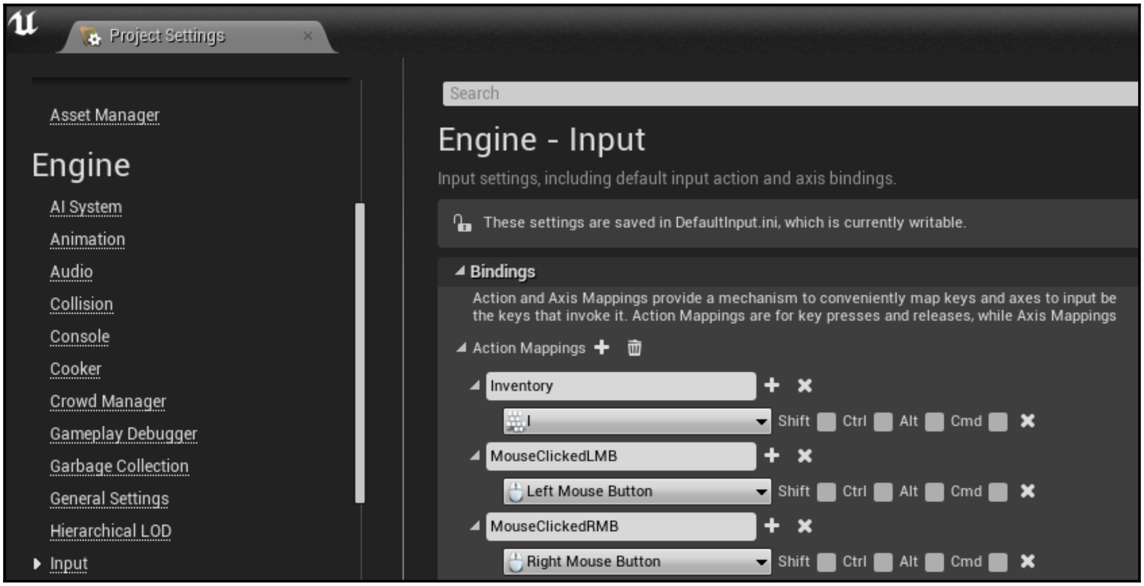
This is very similar to our left mouse click function. We simply check the click position against all the widgets. If any `Widget` was hit by the right-click, and that `Widget` has a `Spell` object associated with it, then a spell will be cast by calling the avatar's `CastSpell()` method.

Activating right mouse button clicks

To connect this HUD function to run, we need to attach an event handler to the mouse right-click. We can do so by performing the following steps:

1. Go to **Settings** | **Project Settings**; the dialog pops up

2. Under **Engine - Input,** add an action mapping for **Right Mouse Button**, as shown in the following screenshot:



3. Declare a function in `Avatar.h/Avatar.cpp` called `MouseRightClicked()` with the following code:

```
void AAvatar::MouseRightClicked()
{
  if( inventoryShowing )
  {
    APlayerController* PController = GetWorld()-
     >GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
    hud->MouseRightClicked();
  }
}
```

4. Then, in `AAvatar::SetupPlayerInputComponent()`, we should attach the `MouseClickedRMB` event to that `MouseRightClicked()` function:

```
// In AAvatar::SetupPlayerInputComponent():
PlayerInputComponent->BindAction("MouseClickedRMB", IE_Pressed,
this,
        &amp;AAvatar::MouseRightClicked);
```

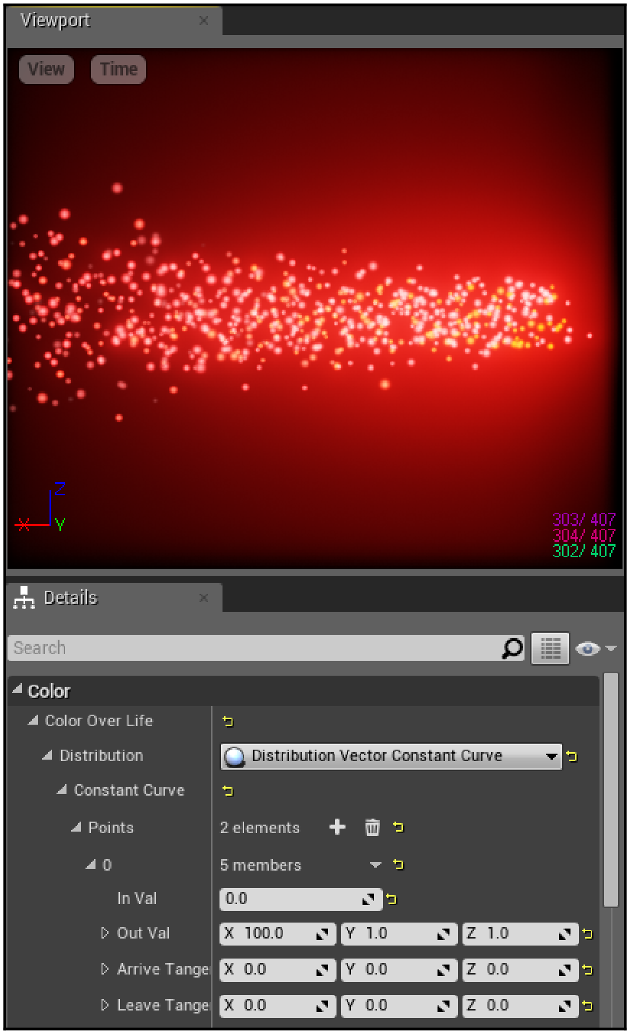
We have finally hooked up spell casting. Try it out; the gameplay is pretty cool, as shown in the following screenshot:

Creating other spells

By playing around with particle systems, you can create a variety of different spells that do different effects. You can create spells for fire, or lightning, or to push the enemy away from you. You've probably run into many other possible spells while playing other games.

The fire spell

You can easily create a fire variant of our blizzard spell by changing the color of the particle system to red. This is how the fire variant of our blizzard spell will appear:



The out val of the color changed to red

Exercises

Try the following exercises:

- **Lightning spell**: Create a lightning spell by using the beam particle. Follow Zak's tutorial for an example of how beams are created and shot in a direction, at `https://www.youtube.com/watch?v=ywd3lFOuMV8&list=PLZlv_N0_O1gYDLyB3LVfjYIcbBe8NqR8t&index=7`.
- **Forcefield spell**: A forcefield will deflect attacks. It is essential for any player. Suggested implementation: derive a subclass of `ASpell` called `ASpellForceField`. Add a bounding sphere to the class, and use that in the `ASpellForceField::Tick()` function to push the monsters out.

Summary

You now know how to create spells to defend yourself in game. We've used particle systems to create a visible spell effect, and an area that can be used to cause damage to any enemies inside it. You can expand on what you've learned to create even more.

In the next chapter we will look into a newer and easier way to build the user interface.

14
Improving UI Feedback with UMG and Audio

User feedback is very important in games, because the user needs information about what is going on in the game (score, hp, displaying inventory, and so on). In previous chapters, we've created a very simple HUD to display text and items in your inventory, but if you want a game that looks professional you'll want to have a much nicer **User Interface** (**UI**) than that!

Fortunately, there are easier ways of building a UI now with Unreal Motion Graphics UI Designer (UMG), a system included with UE4, just for this purpose. This chapter will show you how to use it to take what we've done before and make something that looks much better and has more functionality. We'll start updating the inventory window, and I will make suggestions on how you can continue the process and update the rest of the UI.

Another way of providing feedback is through audio, either in the game itself or through the UI when you interact with it, so we will also be introducing how to play sounds.

The topics we'll be covering are as follows:

- What is UMG?
- Updating the inventory window
- Laying out your UI
- Updating your HUD and adding health bars
- Playing audio

What is UMG?

You may have noticed that the code we've used to draw on the screen has been very complicated. Every element needs to be placed onscreen manually. You might ask yourself if there's an easier way. And there is! It's the Unreal Motion Graphics UI Designer, or UMG.

UMG simplifies the process of creating a UI by using special blueprints to allow you to lay out the interface visually. This can also let you have a tech-savvy artist do the layout for you, while you hook everything up. We will be using this, but since this is a C++ book we will be handling most of the behind the scenes functionality in C++.

In order to use UMG, first you need for find the `GoldenEgg.Build.cs` file in your Visual Studio project. `.cs` files are generally C#, not C++, but you don't have to worry about that since we'll only be making minor changes to this file. Find this line:

```
PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject",
"Engine", "InputCore" });
```

And add the following to that list:

```
, "UMG", "Slate", "SlateCore"
```

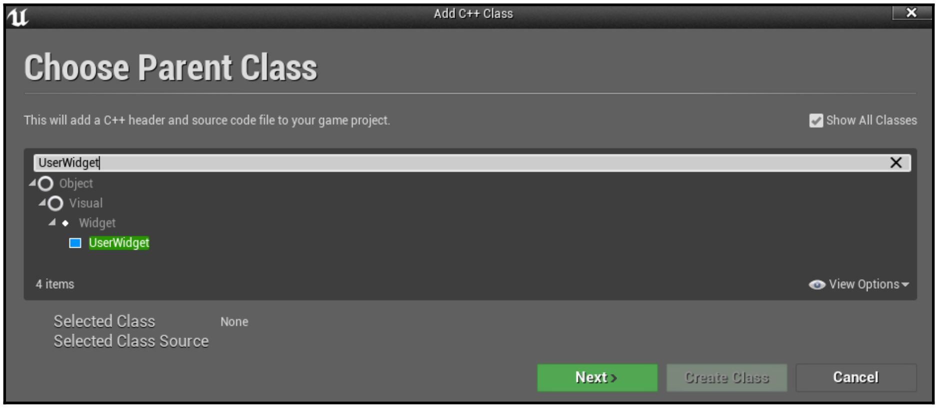
You may want to restart the engine once you do that. Then you'll be ready to code in UMG!

Updating the inventory window

We're going to start by updating the inventory window. What we have right now isn't a real window, just images and text drawn on the screen, but now you'll see how you can easily create something that looks more like a real window—with a background and a **Close** button, and the code will be much simpler.

The WidgetBase class

To create a C++ class for a UMG Widget, you need to create a new class based on `UserWidget`. To find it when adding a new C++ class, you need to check **Show All Classes** and search for it:



Name your class `WidgetBase`. This will be the base class from which you will derive any other Widget classes you create. This allows you to put functionality in this class that will be reused in many different Widgets. In this case, I put the functionality for `CloseButton` in there. Not all Widgets will need one, but if you're trying for a standard window it is generally a good idea.

Here is the code for `WidgetBase.h`:

```
#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "UMG/Public/Components/Button.h"
#include "WidgetBase.generated.h"

/**
 * WidgetBase.h
 */
UCLASS()
class GOLDENEGG_API UWidgetBase : public UUserWidget
{
    GENERATED_BODY()
public:
    UPROPERTY(meta = (BindWidgetOptional))
    UButton* CloseButton;

    bool isOpen;
    bool Initialize();
    void NativeConstruct();

    UFUNCTION(BlueprintCallable)
    void CloseWindow();
};
```

This sets up all the code that allows you to use a button to close the window. `CloseButton` will be the name of the button we create in the design Blueprint.

The line `UPROPERTY(meta = (BindWidgetOptional))` should automatically link the `CloseWindow` variable to the `Button` object with the same name in the blueprint we will create in a little while. If you know the widget will always be there you can use `UPROPERTY(meta = (BindWidget))` instead, but in this case there may be cases where there is no button needed to close a window.

And here is `WidgetBase.cpp`:

```
#include "WidgetBase.h"
#include "Avatar.h"
#include "Kismet/GameplayStatics.h"

bool UWidgetBase::Initialize()
{
    bool success = Super::Initialize();
    if (!success)  return false;

    if (CloseButton != NULL)
    {
        CloseButton->OnClicked.AddDynamic(this, &UWidgetBase::CloseWindow);
    }
    return true;
}

void UWidgetBase::NativeConstruct()
{
    isOpen = true;
}

void UWidgetBase::CloseWindow()
{
    if (isOpen)
    {
        AAvatar *avatar = Cast<AAvatar>(
            UGameplayStatics::GetPlayerPawn(GetWorld(), 0));

        avatar->ToggleInventory();
        isOpen = false;
    }
}
```

> **TIP:** If the UMG includes in this chapter don't work for you you might need to add `Runtime/` to the front of the path. But they should work like this (and do work in my project).

The following line is what sets the `OnClicked` event to call a specific function:

```
CloseButton->OnClicked.AddDynamic(this, &UWidgetBase::CloseWindow);
```

We no longer need to set up everything in the input settings like we did previously, since UMG buttons already are set up to handle `OnClicked`, and you just need to tell it what function to call. If for some reason that doesn't work, I'll show you how to work around it by setting up `OnClicked` in the blueprint later. Since `CloseButton` is optional, you do need to check it to make sure it's not set to `NULL` to avoid errors.

The `isOpen` variable is there to handle the common UI issue where sometimes clicks (or key presses) register multiple times, causing the function to be called more than once, which could cause errors. By setting `isOpen` to true the first time you call the `OnClicked` function you are making sure it doesn't run it more than once, since it will only run if the value is false. Of course, you also need to make sure the value is reset if you reopen the window, which is where the `NativeConstruct()` function comes in.

The InventoryWidget class

Now you will want to create the specialized class for handling the inventory widget, derived from `WidgetBase`. If for some reason you can't find `WidgetBase` to create the class the usual way, uncheck **Actors Only** under **Filters**. Call this one `InventoryWidget`.

Once you've created that class you can start adding the code. First, here's `InventoryWidget.h`:

```
#include "CoreMinimal.h"
#include "WidgetBase.h"
#include "UMG/Public/Components/Image.h"
#include "UMG/Public/Components/TextBlock.h"
#include "UMG/Public/Components/Button.h"
#include "InventoryWidget.generated.h"

/**
 *
 */
UCLASS()
class GOLDENEGG_API UInventoryWidget : public UWidgetBase
```

```
{
    GENERATED_BODY()

public:
    const int kNumWidgets = 2;
    //image widgets
    UPROPERTY(meta = (BindWidget))
        UImage* InventoryImage1;

    UPROPERTY(meta = (BindWidget))
        UImage* InventoryImage2;

    //text widgets
    UPROPERTY(meta = (BindWidget))
        UTextBlock* InventoryText1;

    UPROPERTY(meta = (BindWidget))
        UTextBlock* InventoryText2;

    //Invisible Buttons
    UPROPERTY(meta = (BindWidget))
        UButton* InventoryButton1;

    UPROPERTY(meta = (BindWidget))
        UButton* InventoryButton2;

    bool Initialize();

    void HideWidgets();
    void AddWidget(int idx, FString name, UTexture2D* img);

    UFUNCTION(BlueprintCallable)
    void MouseClicked1();
    UFUNCTION(BlueprintCallable)
    void MouseClicked2();
};
```

This file is much more complicated. We are again using `BindWidget` to set up objects in the Blueprint. While you could lay out widgets in code like we did previously (but you should be able to create a subwidget including the image, text, and button), to keep things simpler I just laid out two on screen and referenced them separately. You can always add more yourself later for practice.

So, in this particular case we have widgets set up for two images, two text blocks, and two buttons. There is an `Initialize` function to set those up, as well as functions to add a widget, hide all widgets, and mouse click handlers for each button.

Then we need to write `InventoryWidget.cpp`. First, add the includes at the top of the file:

```
#include "InventoryWidget.h"
#include "MyHUD.h"
#include "Runtime/UMG/Public/Components/SlateWrapperTypes.h"
```

Then set up the `Initialize` function:

```
bool UInventoryWidget::Initialize()
{
    bool success = Super::Initialize();
    if (!success)  return false;

    if (InventoryButton1 != NULL)
    {
        InventoryButton1->OnClicked.AddDynamic(this,
&UInventoryWidget::MouseClicked1);
    }
    if (InventoryButton2 != NULL)
    {
        InventoryButton2->OnClicked.AddDynamic(this,
&UInventoryWidget::MouseClicked2);
    }
    return true;
}
```

This function sets up the `OnClicked` functions for the buttons. Then add the functions to handle Widgets:

```
void UInventoryWidget::HideWidgets()
{
    InventoryImage1->SetVisibility(ESlateVisibility::Hidden);
    InventoryText1->SetVisibility(ESlateVisibility::Hidden);
    InventoryImage2->SetVisibility(ESlateVisibility::Hidden);
    InventoryText2->SetVisibility(ESlateVisibility::Hidden);
}

void UInventoryWidget::AddWidget(int idx, FString name, UTexture2D* img)
{
    if (idx < kNumWidgets)
    {
        switch (idx)
        {
        case 0:
            InventoryImage1->SetBrushFromTexture(img);
            InventoryText1->SetText(FText::FromString(name));
            InventoryImage1->SetVisibility(ESlateVisibility::Visible);
            InventoryText1->SetVisibility(ESlateVisibility::Visible);
```

```
            break;
        case 1:
            InventoryImage2->SetBrushFromTexture(img);
            InventoryText2->SetText(FText::FromString(name));
            InventoryImage2->SetVisibility(ESlateVisibility::Visible);
            InventoryText2->SetVisibility(ESlateVisibility::Visible);
            break;
        }

    }
}
```

`HideWidgets` hides all the Widgets in the window so they won't show up at all if there is nothing there. `AddWidget` takes an index, a name, and a texture for the image itself, then sets up the Widgets for that index. The **Text Widget** has a `SetText` function that lets you pass in `FText` (`FText::FromString` converts it from `FString` to `FText`). The **Image Widget** has `SetBrushFromTexture` that sets the image.

Finally, you will need to set up the `MouseClicked` functions:

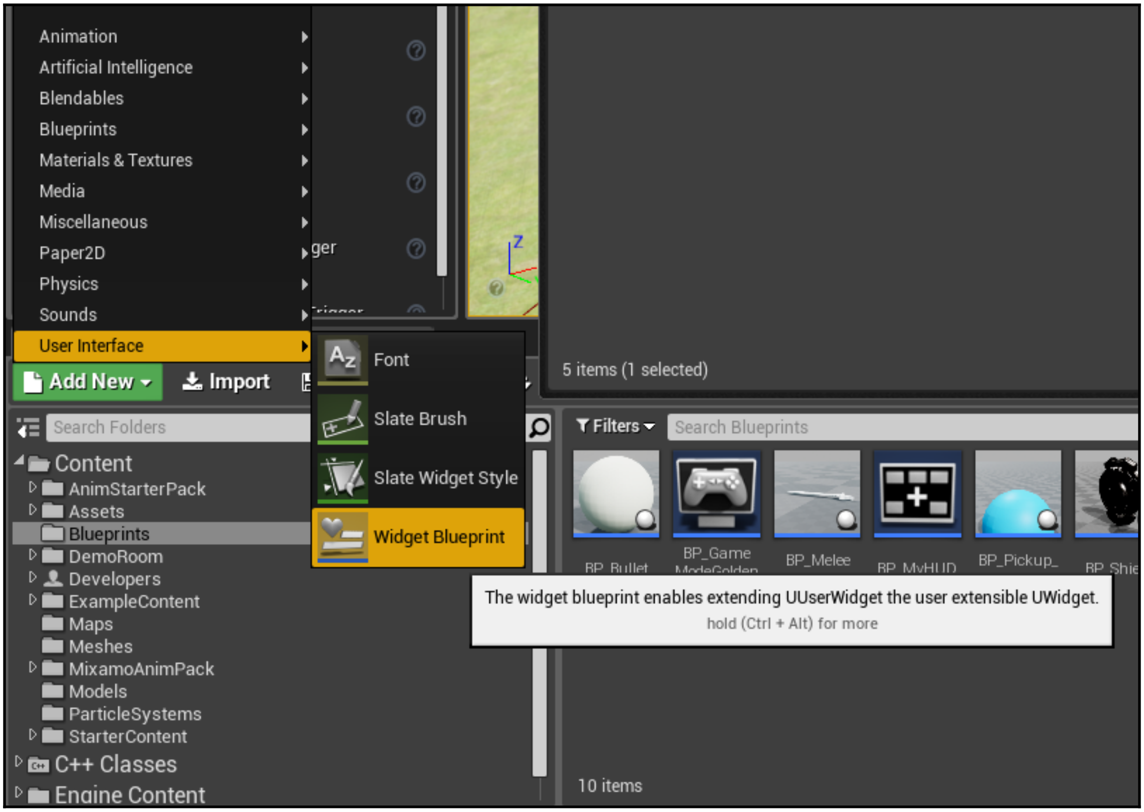
```
void UInventoryWidget::MouseClicked1()
{
    // Get the controller & hud
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());
    hud->MouseClicked(0);
}

void UInventoryWidget::MouseClicked2()
{
    // Get the controller & hud
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());
    hud->MouseClicked(1);
}
```

These just call the HUD's `MouseClicked` function with the button's index (Hint: this won't compile until those HUD functions are updated to take the index). If you want to experiment further, later you can look into another way to get the index based on the button that was clicked, so you can use the same function for all the buttons.

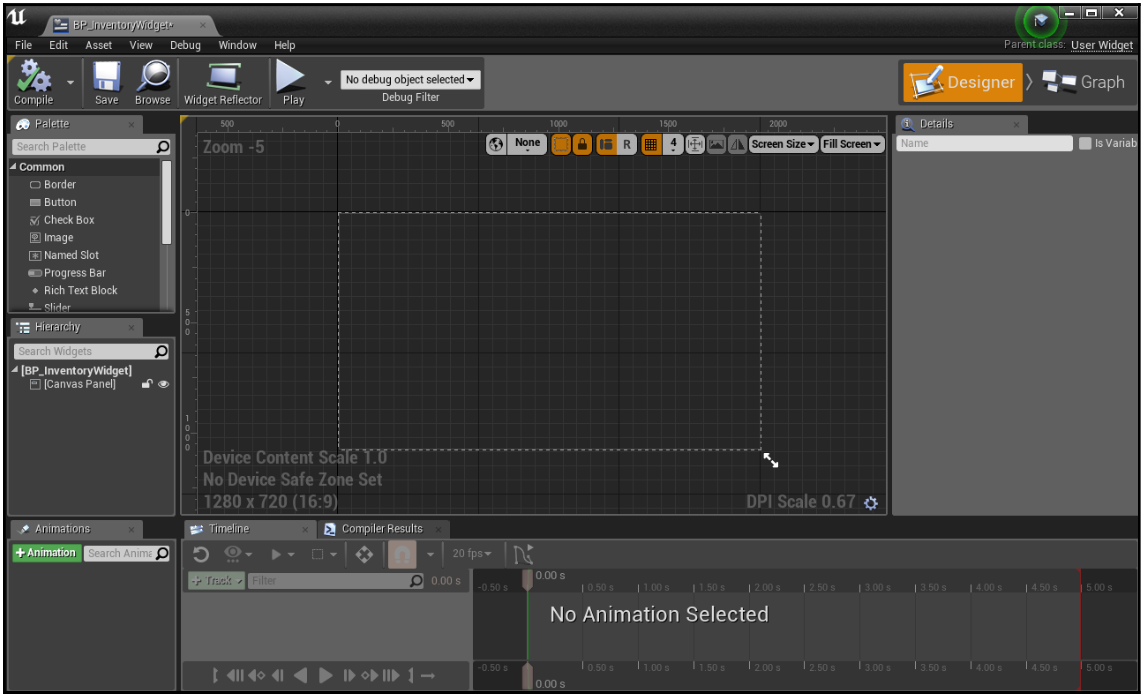
Setting up the widget blueprint

Next, you will need to set up the blueprint. Since this is a special kind of blueprint, setting one up with its own class is a little trickier. You can't just create a blueprint of the class or you won't have a design blueprint. Instead, you have to create the design blueprint first, and then change the parent.

To do this, go into the Content Browser and select the directory you want to put it in, and then select **Add New** | **User Interface** | **Widget Blueprint**:

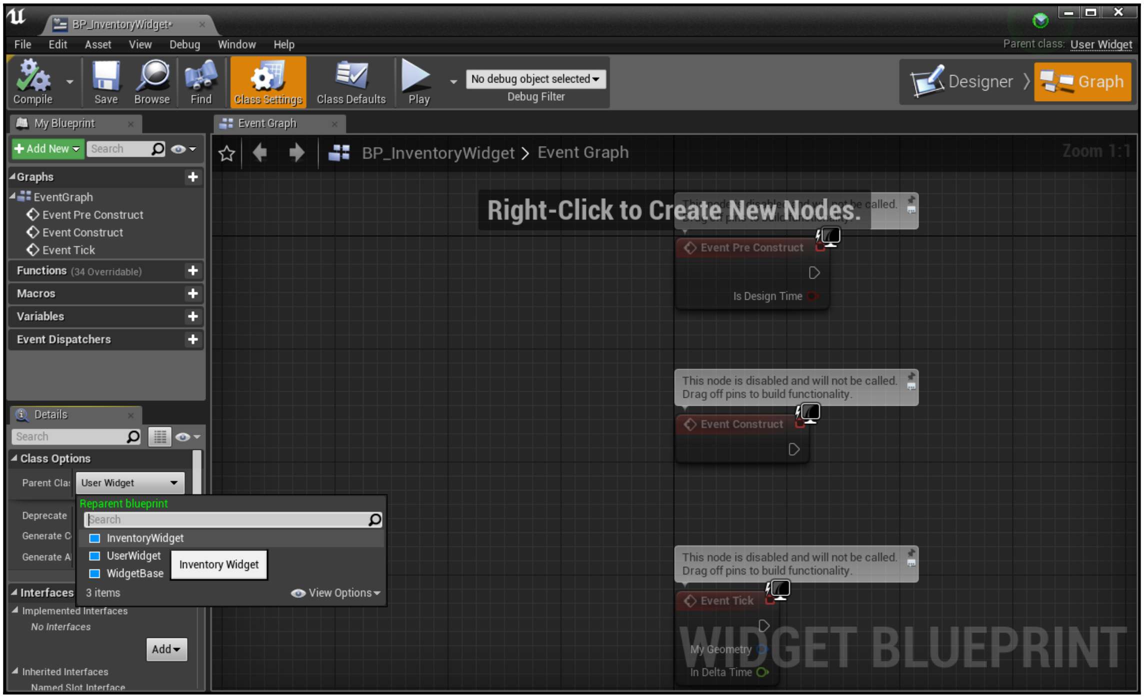


Rename it `BP_InventoryWidget` and then double-click to open it. You should see something like this:



In the center you will visually lay out the screen, and the box represents the edges of the theoretical screen you are aiming for. On the left side the Palette shows you the basic UI objects you can add to the screen. You'll see many common objects such as images, text fields, progress bars, buttons, checkboxes, and sliders. That's a lot of functionality you basically get for free. Once you get to the point of setting up a settings window for your game many of those will come in handy.

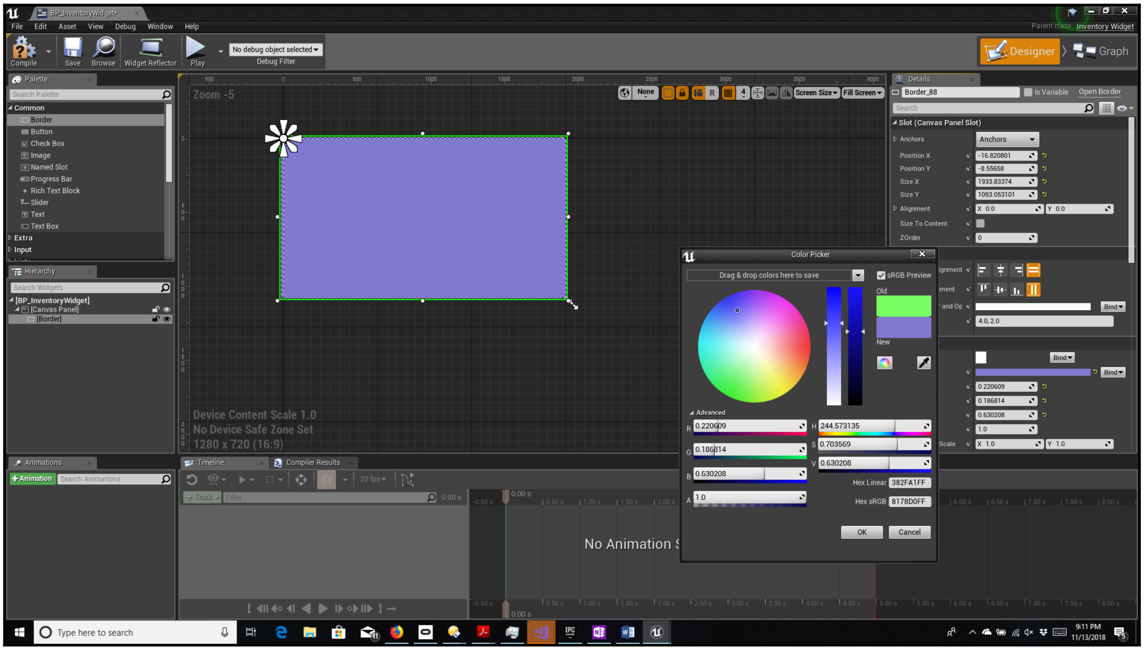
But first, we need to change the parent class on this, which you will do here. Select **Graph** on the top-right corner and **Class Settings** on the top toolbar, then look under **Details** for **Class Options** and choose the dropdown by **Parent Class**. Select **InventoryWidget** there:



Now we'll want to go back to **Designer** and start laying out the screen!

There should already be a **Canvas Panel** on the screen. You can click on the bottom right corner and drag to make it the size you want. The canvas should usually be the size of the full screen. All other UI widgets will go inside the canvas. When you drag this, it will show the various resolutions on screen that you are aiming for. You will want to pick one similar to the resolution you are aiming for.

Then select **Border** under the **Palette** and drag it out onto the screen. This will be the background of the window. You can click the corner and drag it to the size you want. You can also find the color bar on the right (next to **Appearance > Brush Color** under **Details**) and click on it to open a **Color Picker** to choose the color of the background:



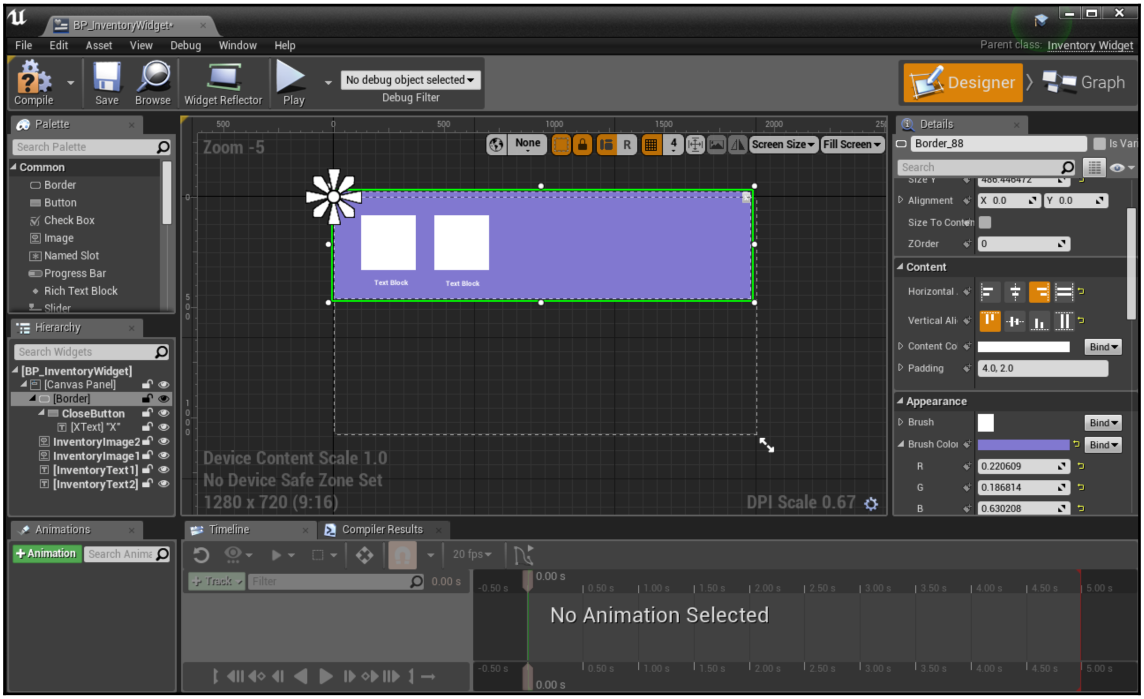
You can also rename the object under **Details**. Once you have that, click and drag a **Button** onscreen and position it on the top-right corner of the background. If it tries to fill the entire **Border** object make sure you select **Canvas Panel** in the hierarchy or drag it outside the **Border** object and then drag it on top of it. Make sure you name this `CloseButton`. You can also put a text object with the letter X in it if you want to make it look more like a close button. You should uncheck **Is Enabled** under **Behavior** in the **Details** so it won't block mouse clicks.

Next, you will want to position two image objects and two text objects (you can add more later). Make sure the names exactly match the names you used in the code or they won't work. In the **Text** fields, you will find it much easier to set a font. Under **Details** | **Appearance** you will find font options like you're used to in any word processor, and you can use fonts already on your computer (although, if you still want to download a font there is nothing stopping you). You can also use the font you added earlier.

Also, for `OnClicked`, you will want to add a button. You could just add one underneath, but I used a common UI method: the invisible button. Drag a button out and have it cover both the image and the text for one of them. Then go into the **Background color** and set the alpha (**A**) to `0`. Alpha is a measurement of how transparent a color is, and `0` means you won't be able to see it at all.

> **TIP:** If you later have trouble clicking the buttons other objects might be in the way. Try dragging them so they are behind the button or look into ways of disabling clicks on those objects.

In the end, you should have something like this:



Also, carefully note the options under **Content** on the right when you have **Border** selected. Here is where you can set **Horizontal** and **Vertical** alignment. Always try to set these, so if you want something to always be positioned in the top-left corner of the screen the alignment will be set to **Horizontally Align Left** and **Vertically Align Top**. If you don't set alignment for every object the results could be unpredictable in different screen resolutions. I'll get into that more later.

But for now, this will be your inventory window. It doesn't have to look exactly like mine, so have fun and experiment with the visual layout! Although, remember that you probably don't want it to take up the entire screen so you can see the spell being cast after you click (although you can look into closing the window when you click on a spell later).

AMyHUD changes

But that's not all! We still have to modify our existing classes to support this new Widget, starting with the `AMyHud` class. To make things simpler we won't be replicating all the previous functionality here. Instead, we will just be setting up the `OnClicked` function to cast spells, since that's going to be a lot more useful in game than dragging items around on the screen. Right-clicks aren't handled automatically by UMG but you can look into that more yourself if you want to add it later, and you can also look into the previous click and drag functionality, so you might want to comment out the older code instead of deleting it if you think you might want it later.

For now, the `MouseMoved` and `MouseRightClicked` functions are gone, and the `MouseClicked` function now takes an `int` index. We also have new functions for `OpenInventory` and `CloseInventory`, so `MyHUD.h` should have this now:

```
    void MouseClicked(int idx);

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Widgets")
        TSubclassOf<class UUserWidget> wInventory;

    UInventoryWidget* Inventory;

    void OpenInventory();
    void CloseInventory();
```

Also add #include "InventoryWidget.h" at the top of the file. Some of the other functions will also be modified. So, now we will look at `AMyHUD.cpp` and you will see just how much simpler the new versions of functions are. Here are the new functions for handling widgets:

```
void AMyHUD::DrawWidgets()
{
    for (int c = 0; c < widgets.Num(); c++)
    {
        Inventory->AddWidget(c, widgets[c].icon.name, widgets[c].icon.tex);
    }
}

void AMyHUD::addWidget(Widget widget)
{
```

```
        widgets.Add(widget);
    }

    void AMyHUD::clearWidgets()
    {
        widgets.Empty();
    }
```

We also need to update the `MouseClicked` function to this:

```
    void AMyHUD::MouseClicked(int idx)
    {
        AAvatar *avatar = Cast<AAvatar>(
            UGameplayStatics::GetPlayerPawn(GetWorld(), 0));
        if (widgets[idx].bpSpell)
        {
            avatar->CastSpell(widgets[idx].bpSpell);
        }
    }
```

This will cast the spell based on the index passed in. Then there are the new functions to open and close the inventory:

```
    void AMyHUD::OpenInventory()
    {
        if (!Inventory)
        {
            Inventory =
    CreateWidget<UInventoryWidget>(GetOwningPlayerController(), wInventory);
        }
        Inventory->AddToViewport();
        Inventory->HideWidgets();
    }

    void AMyHUD::CloseInventory()
    {
        clearWidgets();
        if (Inventory)
        {
            Inventory->HideWidgets();
            Inventory->RemoveFromViewport();
        }
    }
```

The main part is to add or remove the new Widget from the `Viewport`. We also want to hide the widgets visually to keep empty widgets from displaying, and to clear out all the widgets when the window is closed.

We also changed `struct Widget` to remove all the positioning information. Any reference to it should have been removed, but if you get any errors later (you won't be able to compile until you make the changes to the Avatar class) make sure `MouseMoved` and `MouseRightClicked` are gone or commented out, and nothing else is referencing them. The newer, simpler widget should look like this:

```
struct Widget
{
    Icon icon;
    // bpSpell is the blueprint of the spell this widget casts
    UClass *bpSpell;
    Widget(Icon iicon)
    {
        icon = iicon;
    }
};
```

AAvatar changes

In `AAvatar` we will primarily be modifying the `ToggleInventory` function. The newer function will look like this:

```
void AAvatar::ToggleInventory()
{
    // Get the controller & hud
    APlayerController* PController =
GetWorld()->GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());

    // If inventory is displayed, undisplay it.
    if (inventoryShowing)
    {
        hud->CloseInventory();
        inventoryShowing = false;
        PController->bShowMouseCursor = false;
        return;
    }

    // Otherwise, display the player's inventory
    inventoryShowing = true;
    PController->bShowMouseCursor = true;
    hud->OpenInventory();
    for (TMap<FString, int>::TIterator it =
        Backpack.CreateIterator(); it; ++it)
    {
```

```
            // Combine string name of the item, with qty eg Cow x 5
            FString fs = it->Key + FString::Printf(TEXT(" x %d"), it->Value);
            UTexture2D* tex;
            if (Icons.Find(it->Key))
            {
                tex = Icons[it->Key];
                Widget w(Icon(fs, tex));
                w.bpSpell = Spells[it->Key];
                hud->addWidget(w);
            }
        }
        hud->DrawWidgets();
    }
```

As you can see, many of the same HUD functions were reused, but the new functions for `OpenInventory` and `CloseInventory` are now called from here, so the HUD can display the window before adding the widgets, and remove the window to close it.

Also, delete the following lines from both the `Yaw` and `Pitch` functions:

```
            AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());
            hud->MouseMoved();
```

Also delete the following lines from `MouseRightClicked` (or delete the function, but if you do make sure you remove it from `SetupPlayerInputComponent` as well):

```
            AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());
            hud->MouseRightClicked();
```

Finally, remove these lines from `MouseClicked` (since you don't want to accidentally trigger a spell when you click somewhere that's not part of the inventory):

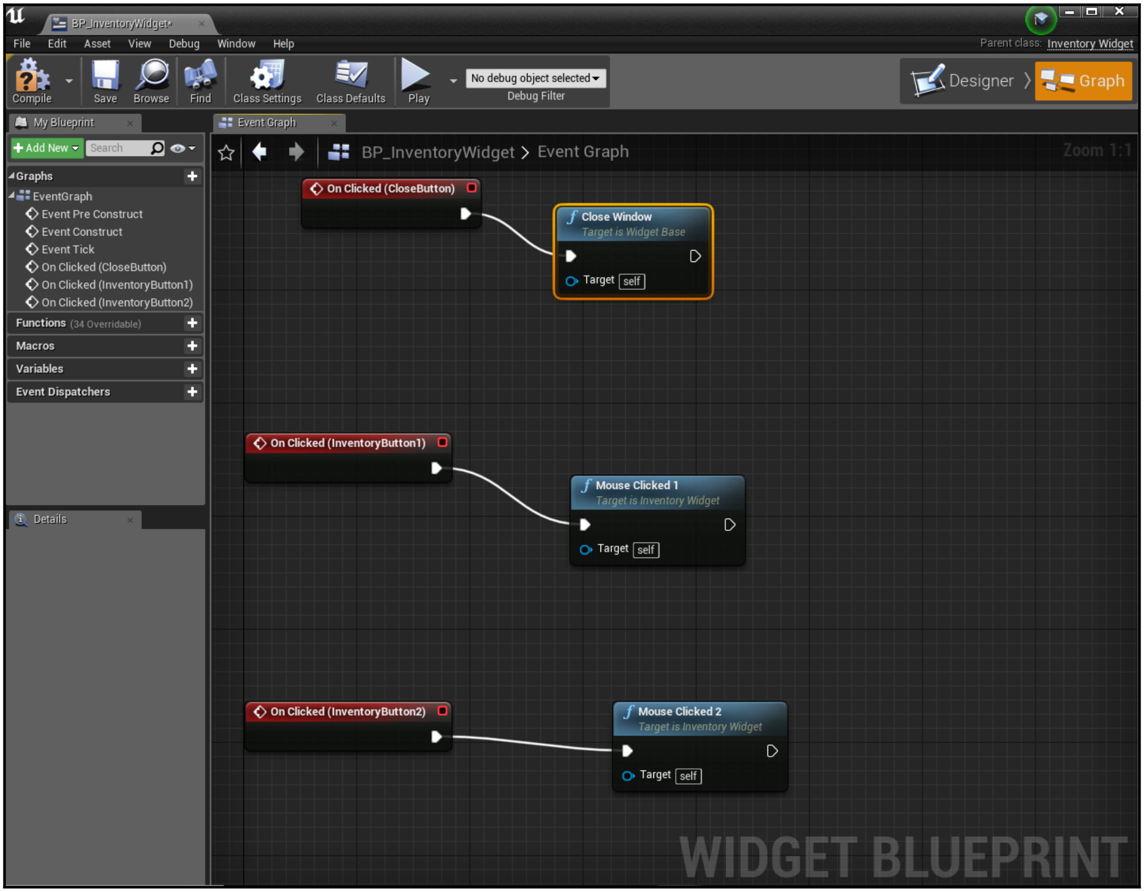
```
        AMyHUD* hud = Cast<AMyHUD>(PController->GetHUD());
        hud->MouseClicked();
```

Now you should be able to compile. Once you do that, go into **BP_MyHUD** and change the **Class Defaults > Widgets > W Inventory** dropdown to **BP_InventoryWidget**.

A note on OnClicked

It's possible your `OnClicked` functions might not work correctly (I ran into that problem myself). If you can't find a solution, you can bypass with with blueprints, which is why I made all the mouse click functions blueprint callable.

If this happens to you, go into the designer for your Widget blueprint and for each button click on it and find **Events** under **Details** and click the green + button next to **On Clicked**. This will add `OnClicked` for that button to the graph and switch to that. You will need to go back to add the other 2 buttons. Then, drag out from the node and add the function you want. It should look something like this:



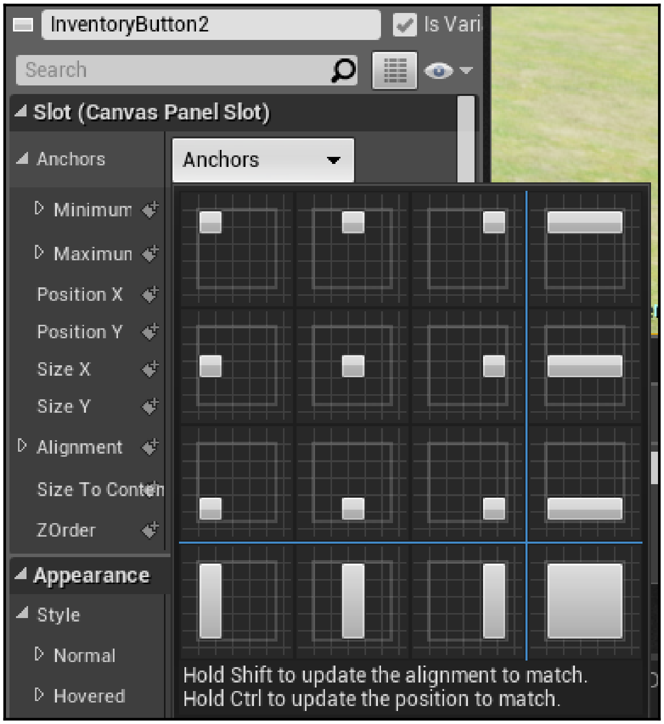
Laying out your UI

When you're laying out a UI, there are important things to keep in mind, and UMG has tools to make this easier for you. One of the most important things to keep in mind is that your game won't always run in the same resolution. If you're doing a mobile game there could be many different devices with different resolutions, and you want your game to look essentially the same on all of them. Even consoles are no longer free from this problem, since both Xbox One and PS4 now have 4K options. So, your game needs to be set up in a way that makes this possible.

If you make all your widgets a specific pixel size and then run it in a resolution much bigger, it could be so small it seems unreadable and buttons could be difficult to click on. In a smaller resolution it could be too big to fit on screen. So, keep this in mind.

The **Canvas Panel** you set up earlier will show you visually how it will look in the size you are aiming for. But for variations on the size you need to keep several things in mind.

First of all, always use anchors. Under details you will see a dropdown list for Anchors. When you open it, you should see something like this:



The nine options on the top-left corner of the blue lines are to align objects. The rows align to the top, middle, and bottom of the screen, while the columns align to the left, middle, and right. So, if you have something you always want to appear at the top-left corner of the screen (like a score, or health bars) you will choose the option on the top-left corner. If you want something else to be centered both horizontally and vertically choose the second row, second column. The little white square basically shows you the positioning.

The remaining options give you ways to have something stretch across the entire screen (no matter what size it is). So, if you want something stretched horizontally on the top, middle, or bottom, look at the right column. For vertical, look at the bottom row. And the one in the bottom-right corner is great if you want a window to stretch across the entire screen.

You can also add a **Scale Box** from the palette if you want everything inside it to scale to fit the screen size. Although if you have something you want to remain a fixed size, like an image, you can check **Size to Content** to prevent it from automatically resizing.

If you want to get more advanced, you can add code to check the screen size and swap out parts of, or the entire, UI, but that's out of the scope of this book, so just keep it in mind if you want to try it later on your own!

TIP

Another important things to keep in mind with your UI is localization. If you want to release your game anywhere outside your own country you will need to localize. This means you will have to get used to not just hardcoding text but use the built-in localization system to add string ids you've set up instead of hardcoding the text. The code will look for specific ids and swap them for the appropriate localized text. You can look into the built-in localization system here: `https://docs.unrealengine.com/en-us/Gameplay/Localization`.

This will also affect how you lay out your UI. The first time you localize your game in German, you'll find out that everything is twice as long! While you may be able to get your translators to come up with shorter ways to say the same thing, you will probably want to make text blocks longer than you think they need to be, or consider finding ways to make the text shrink to fit or scroll.

Updating your HUD and adding health bars

I won't be giving full instructions here, but here are some hints on updating your HUD. Once you do this, it will simplify your code even more!

Creating a HUD class

You will need to create a new class deriving from **WidgetBase** for your new HUD. In this case, you will need the **Canvas Panel** but no background. Make sure everything will stretch across the entire screen.

You will want to keep most of your UI in the corners, so you can add a **Progress Bar** widget to the top-left corner of the screen to display health. Also, consider adding a **Text** widget to tell what it is and/or put the actual numbers onscreen.

For the messages, you can align **Text** widget to the top-middle of the screen and use those to display the text.

Adding health bars

If you've added the recommended **Progress Bar** widget, you'll find drawing health bars to be a lot easier now. You will need to get a reference to it just like you did with the other widgets. Then, all you need to do is call `SetPercent` to show the current health (and reset it whenever it changes).

You no longer have to draw the whole thing yourself, but you can use `SetFillColorAndOpacity` to customize how it looks!

Playing audio

We're going to go back to your code to do one last thing that really helps your game's feedback, yet it somehow tends to be one of the last things anyone thinks about when creating a game: audio.

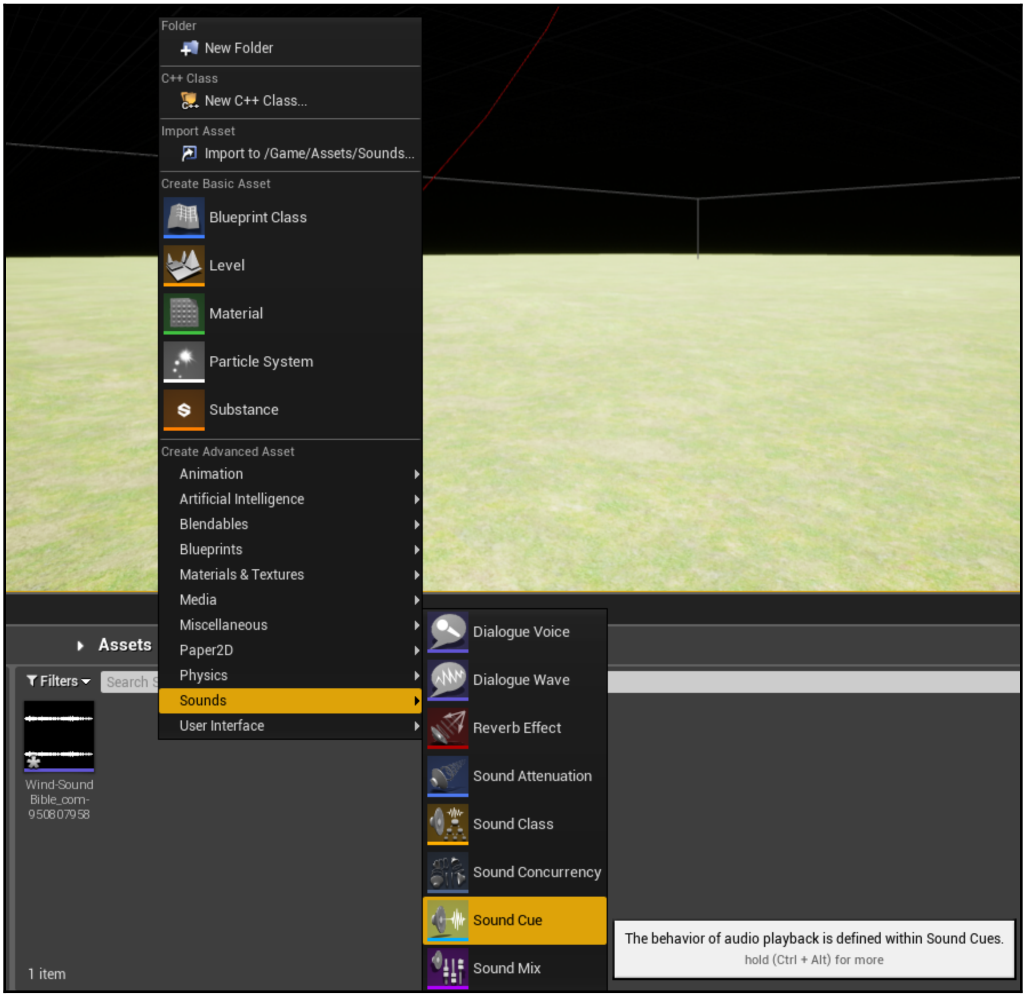
Audio can really enhance your game, from playing a sound when you click a button to adding sound effects, dialog, background music, and ambience. If you're walking alone in the woods at night, the sounds of crickets chirping, your own footsteps, and ominous music can really set the mood. Or, you can have bird sounds and happy music for a completely different mood. It's all up to you!

We're just going to add a sound when you cast your blizzard spell. So look for a free wind sound. There are plenty of sites that offer royalty-free sound files. Some of them want you to mention them in your credits if you use them. For this, I found a public domain sound on a site called `SoundBible.com`, which means anyone can use it. But look for one you like.

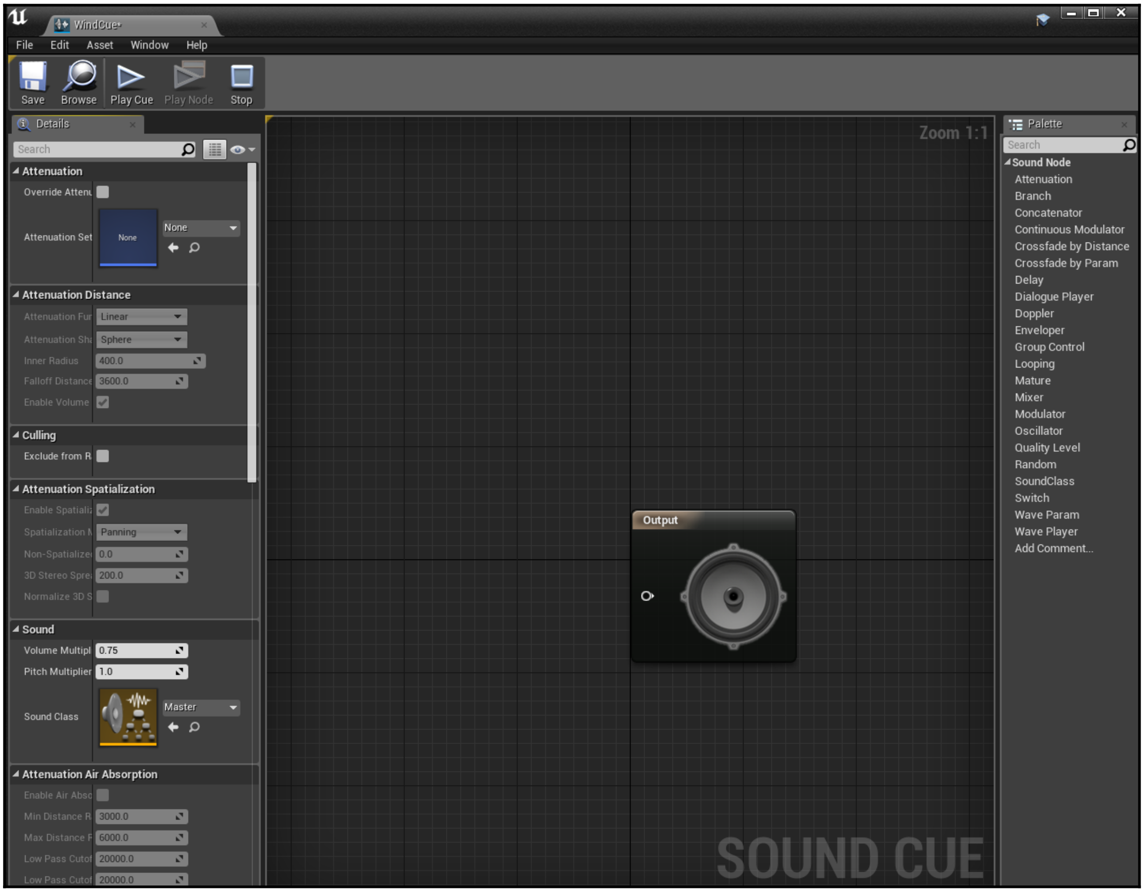
Some sites might make you register to download the sounds. You can even record one yourself if you're feeling ambitious!

> **TIP** I used a .wav file, a standard format, although other formats will probably work. But for small sounds, you may want to stick to .wav because MP3s use compression, which might slow down your game slightly because it needs to de-compress it.

Once you have a file you like, create a folder for **Sounds** and drag your sound file into it from your file manager. Then right-click in the same folder and select **Sounds** | **Sound Cue**:

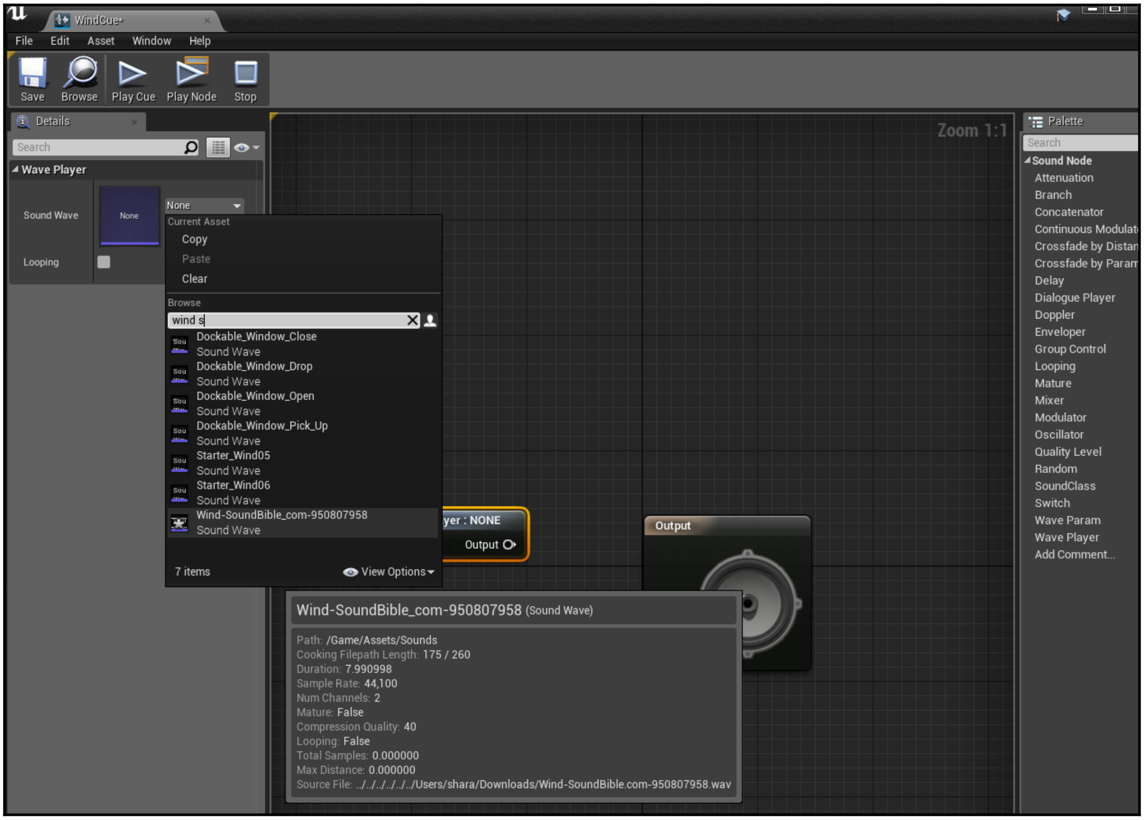


Rename that to **WindCue** and double-click on it to open it in the Blueprint editor. It should look something like this:

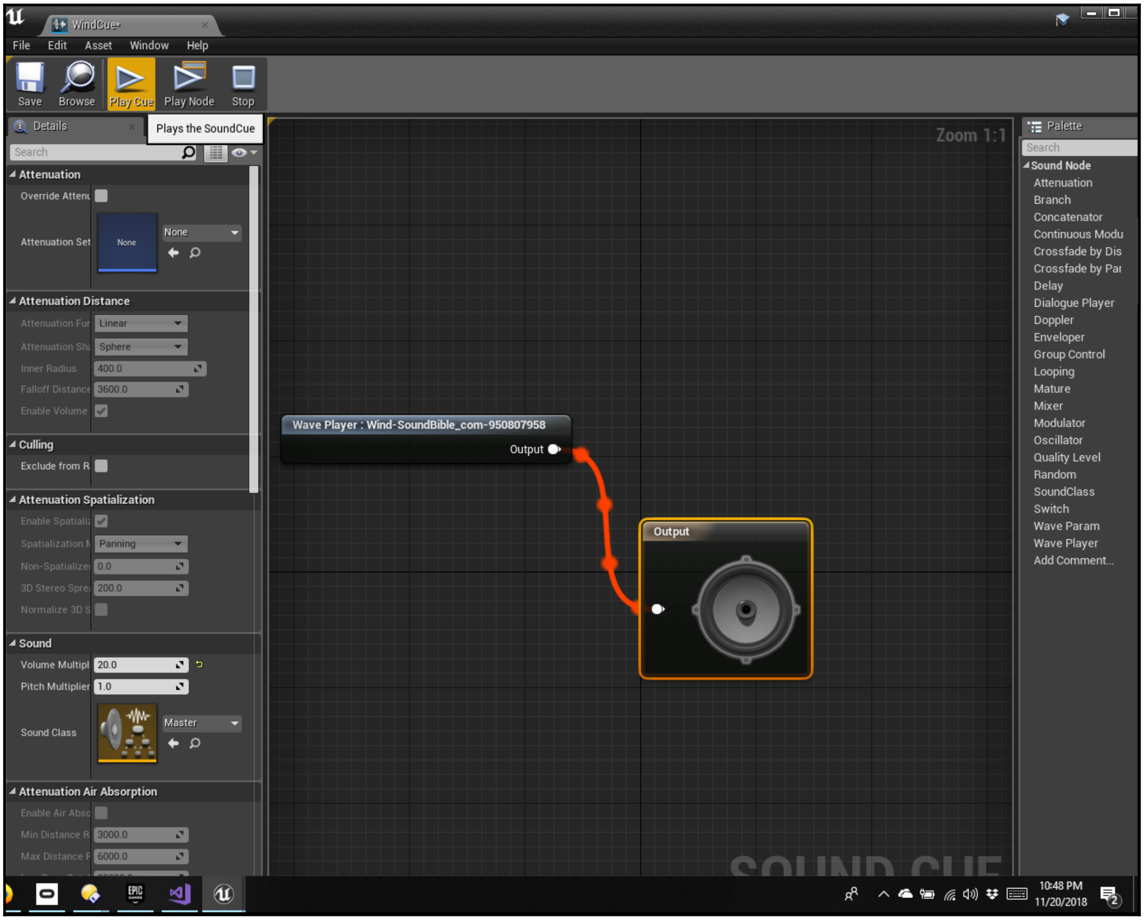


The **SoundCue** is where we will set up our sound. First, right-click anywhere and select **Wave Player** to add one:

Then, select the **Wave Player**. In the details, you will see an option for **Sound Wave**. Select the dropdown list and search for the `.wav` file you added to select it:



Then, click and drag from the output of your **Wave Player** into the **Output** (with the small speaker image). This will hook it up. To test it, you can select **Play Cue** and you should hear it and see the line light up orange as the sound is transferred to the **Output**:



If you don't like the way it sounds, experiment with the options under details. The one I used was too quiet for what I wanted, so I increased the **Volume Multiplier** to make it much louder.

Now that we have the sound set up, it's time to add it to the code. In this case, we'll be updating the `AMyHUD` class. First, add the following line to the top of `MyHUD.h`:

```
#include "Sound/SoundCue.h"
```

Also, add the following in the same file:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Sound")
    USoundCue* audioCue;
```

You will want to store the `SoundCue` reference in the blueprint to make it easy to update.

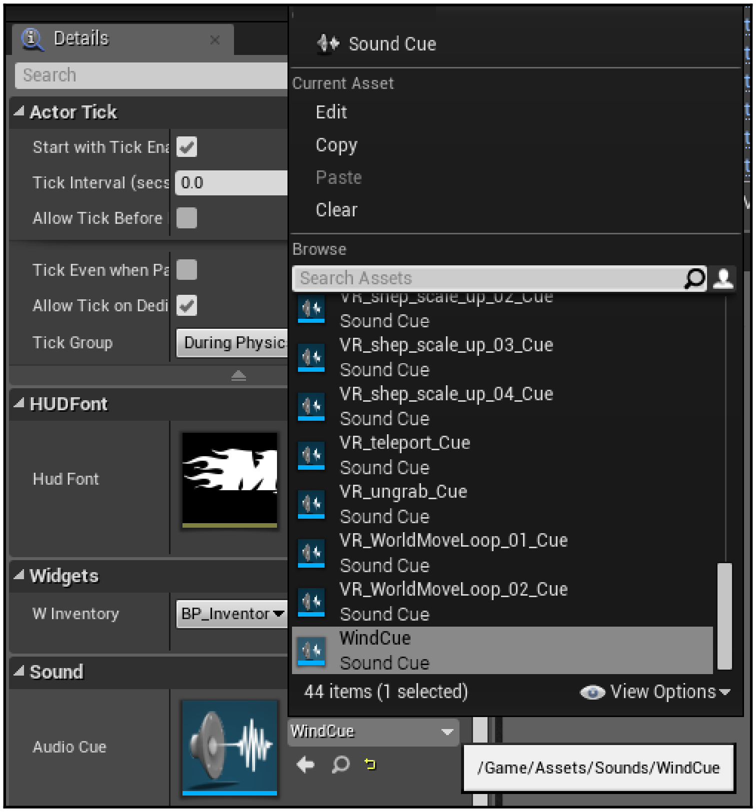
Now, go to `MyHUD.cpp` and add the following line to your `MouseClicked` function right after the call to `CastSpell`:

```
            UGameplayStatics::PlaySound2D(this, audioCue);
```

This will actually play the sound. Make sure you have `#include "Kismet/GameplayStatics.h"` in that file for it to work. For this case, since it is right by the player whenever you cast it, a 2D sound will be fine. If you want things in your environment (like the monsters) to make their own sounds, you will want to look into 3D sounds. UE4 will let you do that!

Now, go back into the editor and compile everything, then go back into the HUD blueprint. You will need to add the `SoundCue` you created to the blueprint.

You can select it from the dropdown list and search for it like this:



Now, save and compile and run the game. Run around until you pick up a **Blizzard spell** and hit *I* to open the inventory. Click on the **Blizzard spell**. You should not only see the spell cast, but you should hear it too!

Summary

You've now gotten a good look at how User Interfaces are created with UMG, and how audio can be added to enhance your experience even further! There is still a lot of work to do, but consider that practice!

We're done with the code for this, but not with the book. Next, we'll be looking at how to take what we have and view it in virtual reality! I'll give you a few hints on that, and then we'll finish things off with an overview of some other advanced features in UE4.

15
Virtual Reality and Beyond

Unless you've been living in a cave, you've probably heard about **Virtual Reality** (**VR**). VR is one of the hottest trends in gaming right now, along with **Augmented Reality** (**AR**), which will be covered later in this chapter. And thanks to such innovations as the ultra-cheap Google Cardboard and similar devices that let you view basic VR on recent smartphones, it's pretty easy to get access to VR technology.

Whether all you have is a Google Cardboard, or you have a higher-end device, such as the Oculus Rift or HTC VIVE, UE4 makes it easy to program for VR. Of course, if you have the PlayStation VR, you would need to become an official Sony developer to program for that (just like if you were programming anything else for PlayStation), so you probably won't be able to do that unless you're working for a company that's doing a PSVR title.

Here, you'll get an overview of VR and UE4 that should help you get started. Here's what we will be covering:

- Getting ready for VR
- Using VR Preview and VR Mode
- Controls in VR
- Tips on VR development

I'm also going to introduce some more advanced features of UE4. We'll start by looking at the other big, hot technology right now, AR, and move on to other technologies. Here is what we will cover:

- AR
- Procedural programming
- Extending functionality with plugins and add-ons
- Mobile, console, and other platforms

Getting ready for VR

It's an exciting time to be getting in to VR development. Maybe you're trying to get into the latest hot technology. Or maybe, like me, you've been reading about VR for decades in cyberpunk books by authors such as William Gibson, Neal Stephenson, Wilhelmina Baird, and Bruce Bethke, and are excited that it's finally here. In either case, here's how you can prepare yourself for your journey into VR programming.

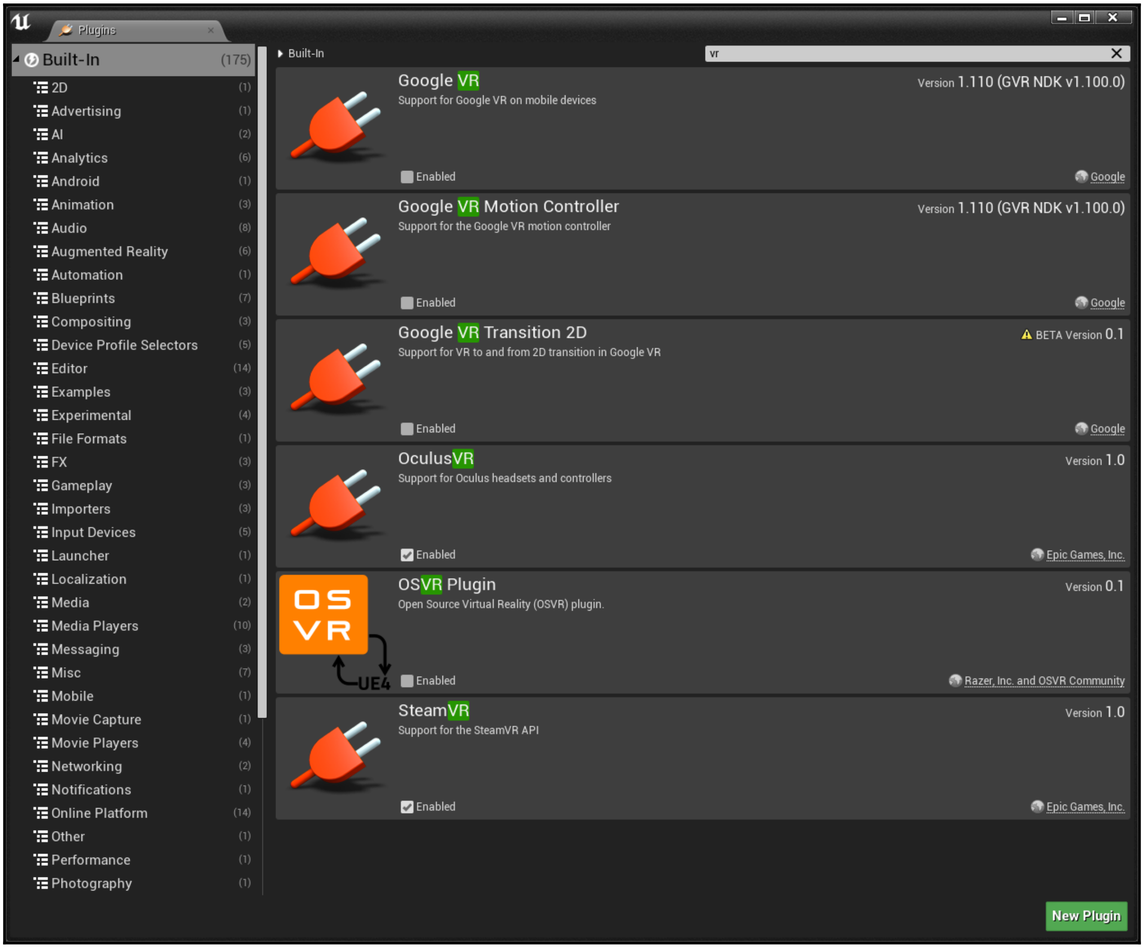
To get started in VR with the Oculus Rift or HTC Vive, first you need a VR-ready computer. Oculus has a free program you can download from their website at `https://ocul.us/compat-tool`, or go to their support page, and it will tell you if there's a problem with your graphics card.

Even if you have a recent computer, unless you specifically got one that was marked as VR-ready, there's a good chance you might need a new graphics card. VR is extremely graphics-intensive, so it requires a pretty high-end (and usually pretty expensive) graphics card.

Of course, if all you want to do is VR on a phone, you may be able to get by without it, but you'll have to do all your testing on the phone and won't have access to a lot of UE4's cool features, such as VR editing.

Once you have a computer that can handle it, you will probably want either an Oculus Rift or an HTC Vive (or both, if you're really serious and have plenty of money to put into it, since neither one is cheap). Whichever device you get will install all the drivers you'll need as part of the setup process.

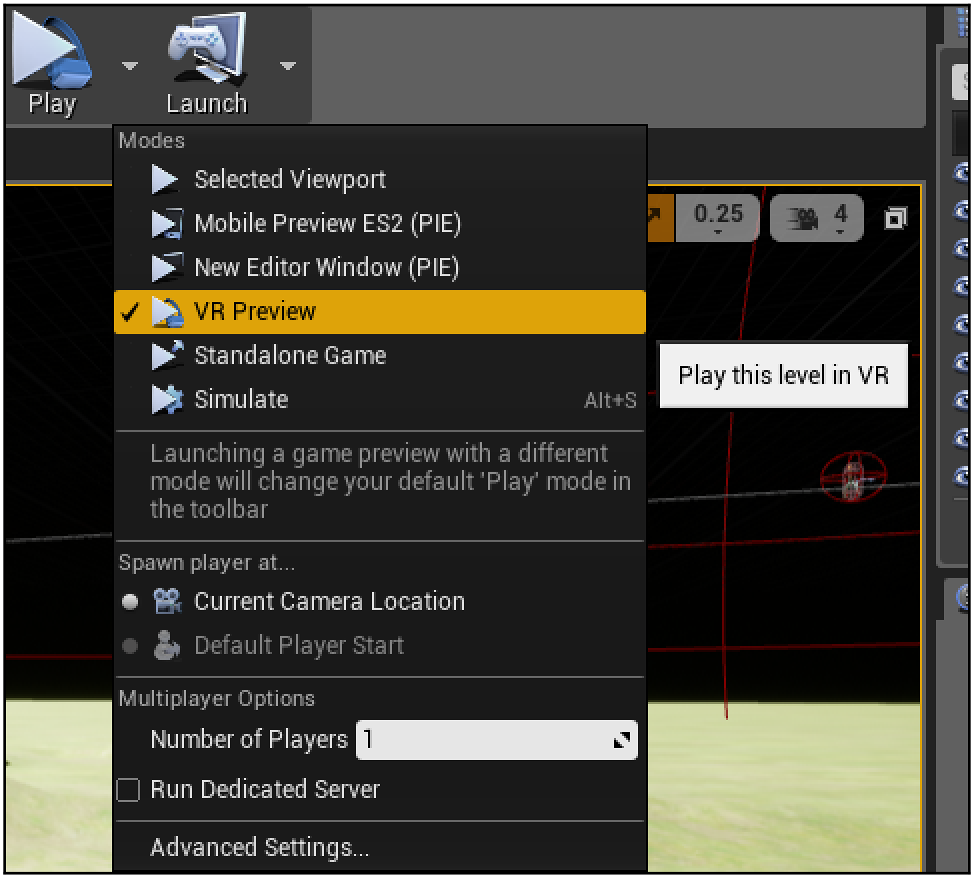
Then, go into UE4, go to **Edit | Plugins**, and make sure you have the plugins for whatever device you own (you can search for them). It should look something like this, depending on your VR hardware:



Also, make sure you have your VR software running (it may start up automatically when you open UE4, depending on your VR hardware).

Using VR Preview and VR Mode

If you want to view something in VR, the good news is you don't need to write anything new! Just go into the existing project, click the arrow next to the **Play** button, and choose **VR Preview:**



Now, just put on your VR headset and you should be able to see the game in VR!

Once you run the game, you can see the game world. You won't be able to move around (you can't see your keyboard or mouse while you're in VR) but you will be able to turn your head to look around you in every direction.

Be very careful if you're prone to motion sickness. This is a serious problem in VR, although there are ways to minimize the effects in your game, which we will talk about later. You might not want to be in VR mode for too long until you get used to it and know how it affects you.

UE4 also has another tool that will really help you out, **VR Mode**. This allows you to actually view and edit the game in VR, so you can see how the changes will look as you make them. This could be very helpful, since many things don't look the same in VR as they do in a non-VR game.

To activate VR Mode either click **VR Mode** in the toolbar or hit *Alt* + *V*:

You can look around, and in VR Mode you will be able to use your motion controllers in the game. You will probably want to look up the controls you need before you go into VR Mode for the first time. There are detailed instructions on VR Mode and the controls you can use within it on the Unreal website at: `https://docs.unrealengine.com/en-us/Engine/Editor/VR`.

If you want to go further, by programming for specific VR systems, such as the Oculus Rift, Vive, Steam VR, or others, there are detailed instructions on working with many different VR systems on the Unreal website. You can find them here: `https://docs.unrealengine.com/en-us/Platforms/VR`.

Controls in VR

You may notice that, in VR mode, the usual controls won't work. You won't even be able to see the keyboard and mouse with a VR headset on, which makes it extremely difficult to use them. Fortunately, higher-end devices have their own controllers available, and UE4 has a **Motion Controller** component you can add to your player pawn, which will allow you to point to things with that instead of a mouse.

> TIP
>
> If you know from the beginning that you are aiming for VR, UE4 has VR-specific classes and templates available that will add some of the functionality you need automatically. There is also a VR expansion plugin that is extremely helpful, and if you're not on a big team of developers, you should really look into it. You can find it here: `https://forums.unrealengine.com/development-discussion/vr-ar-development/89050-vr-expansion-plugin`

UI is very tricky in VR, and many people are still trying to work out the best way of doing it. Your best bet is probably to play a lot of existing games and see what you think works best for you. And make sure you experiment as much as possible, because that's the best way to see what works!

Tips on VR development

VR is a new and exciting technology. People are still figuring out what works, so there is plenty of room for experimentation, and there is plenty of that going on. But you still need to keep some best practices in mind, because you don't want people playing your game to have a bad experience or even to get sick playing your game. If they do, they might not play it again, and would be unlikely to buy your next game. So, you want the experience to be good for everyone.

The biggest problem with VR is simulation sickness (or motion sickness). Some people are affected by this more than others, but if you're not careful, even people who are normally not prone to motion sickness will still have problems. So, it's very important to be careful. And make sure you have other people test your game, because while you might get used to it, that doesn't mean other people won't have trouble.

One of the most important considerations is keeping a very high frame rate. Different devices have different recommendations for minimum frame rates, and if you drop below those, people are likely to start having problems.

Keeping the quality as high as possible is important in general. Anything that looks fake or bad could throw someone out and cause motion sickness. So, if any effects you are trying to achieve don't look as you expected, try doing something else.

You may notice that many VR games don't have the player move around within the game much, if at all, or have them sitting in a moving vehicle. This is another way of avoiding simulation sickness. It's the movement that is the biggest problem, particularly vertical movements such as jumping, or rotating by controller instead of just turning your head. Basically, your mind thinks you're moving, but your body gets conflicting messages because it's not feeling the movement. If you think you're sitting in a car, your body doesn't expect to feel movement, so that's why it seems to work better. Although, if the player is standing while playing, they are likely to have less problems.

There is plenty of information about VR and best practices for it online. The Unreal site has a page about best practices with some very good UE4-specific information at `https://docs.unrealengine.com/en-us/Platforms/VR/ContentSetup`. I recommend going through that before you start your project, because keeping best practices in mind from the beginning is better than finding out at the end of the project that some things don't work or won't work well.

And as I said before, getting people to test it is very important. VR technology is so new that you'll want to make sure it will work for as many people as possible.

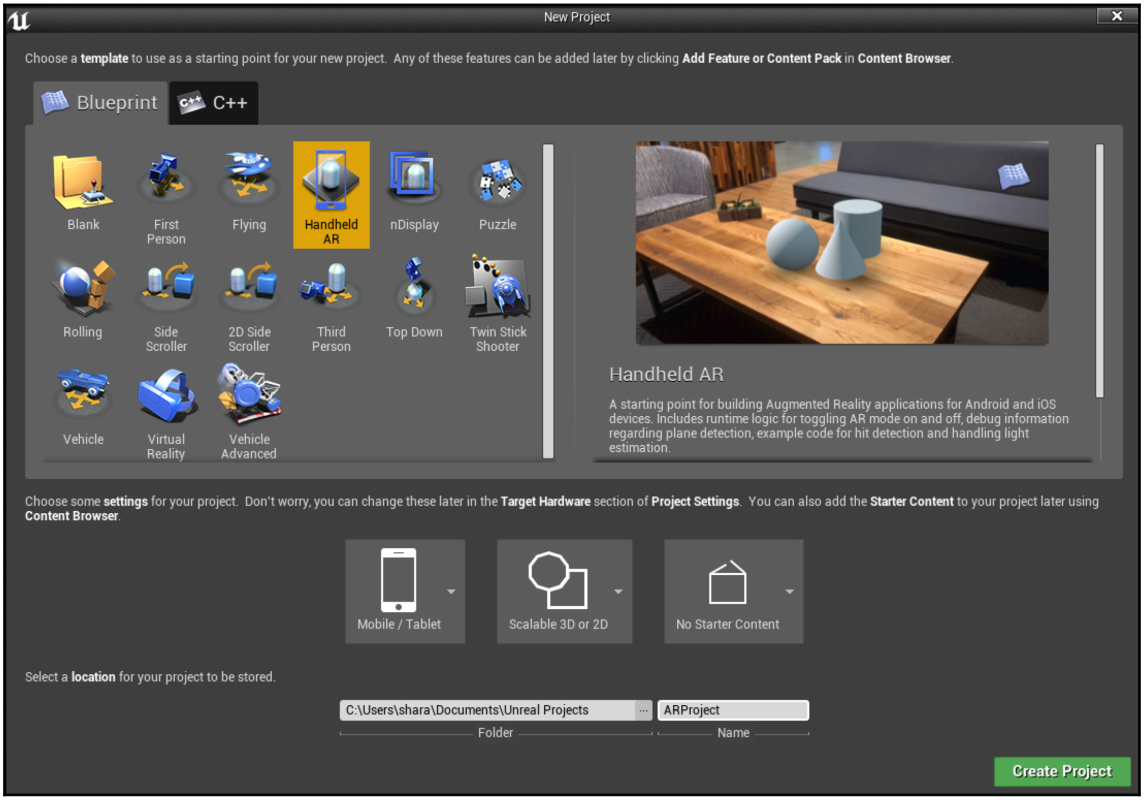
AR

AR is similar to VR, only, instead of being completely surrounded by a virtual world, in this case, you see virtual objects placed in the real world (as viewed through a camera). This could be through a headset, such as Microsoft's HoloLens, or the Magic Leap. But since those are new and only available as expensive devices aimed at developers right now, you will primarily see AR through mobile devices.

Popular AR games on mobile devices include Pokemon Go, where you can catch Pokemon and view them in front of the world around you. In AR mode, you have to look around you until you find a Pokemon (it shows what direction to turn) and catch it. You can even take pictures, which makes for some interesting images. Its precursor, Ingress, let you go to real-world locations in the game, but Pokemon Go really expanded on that.

Due to the success of that game, mobile AR games are very popular right now. Since you're dealing with real-world objects you can't control, this could involve some complex computer vision, but fortunately, UE4 has built-in functionality to help you out.

The two primary mobile AR systems UE4 supports are ARKit for iOS and ARCore for Android. You can find more detailed information about AR programming and the prerequisites for each type on the Unreal site at `https://docs.unrealengine.com/en-us/Platforms/AR`. To start either one, you will want to create a new project using the handheld AR template:



As shown in the preceding screenshot, your settings should be **Mobile / Tablet**, **Scalable 3D or 2D**, and **No Starter Content**. Once you create the project, you can connect your phone to your computer, and if that is fully set up (depending on your phone, you might need to install software on your computer to see it), you should see it under devices when you click the arrow next to **Launch**. Otherwise, you can still use **Mobile Preview ES2 (PIE)** under **Play**.

> While you're not likely to be programming for Magic Leap anytime soon, there is early access documentation on it available on the Unreal site at: `https://docs.unrealengine.com/en-us/Platforms/AR/MagicLeap`.

Procedural programming

Procedural programming in games has been very popular lately. If you've played games such as Minecraft, No Man's Sky, or Spore, you've played a procedural game. The history of procedural games goes back decades, to old text-based games such as Moria, Angband, and NetHack. Rogue like games (named after the original, Rogue) are still a popular genre of game that use procedural techniques to generate random levels, so every time you play, you get a completely different game. So, procedural programming adds replayability that's hard to get when levels have to be built by hand.

Procedural programming lets you create parts of a game, whether it's the environment, levels, or even the audio, through rules and algorithms in code. Basically, instead of having a human being set up every detail, the code does it for you.

The results can be unpredictable, especially in 3D, which is a lot more complicated than drawing out rooms and pathways in 2D text characters. Because of this, sometimes, the procedural levels are created ahead of time, so the designer can choose the ones they like before adding them to the game.

There are many different techniques that help with procedural programming. One is the use of **volumetric pixels** (**voxels**), which let you refer to points in 3D space in a simple way, based on their relation to other voxels. Voxels have been used in many projects, including the now defunct game Landmark (which I worked on), and were supposed to be used in the now canceled EverQuest Next. UE4 supports voxels through plugins, such as Voxel Plugin (`https://voxelplugin.com/`).

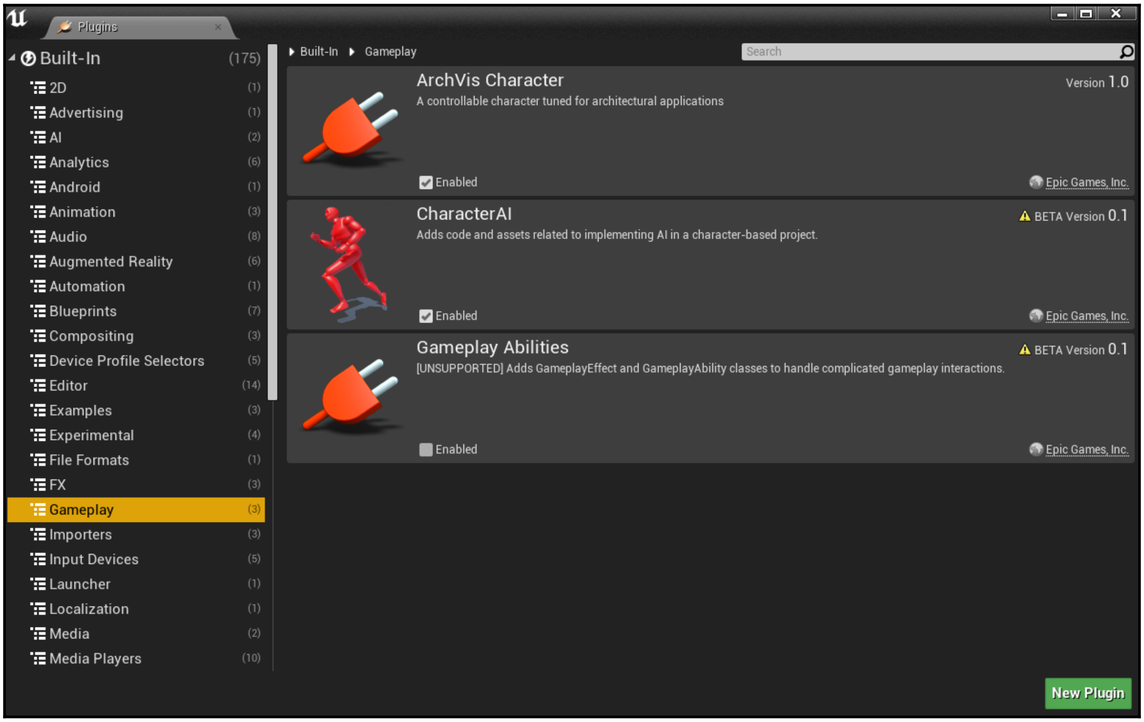
Procedural programming can also be used for music. There are projects that have trained neural networks on specific types of music and come up with some pretty impressive music in similar style. You can also modify the music that plays based on what is happening in the game. Spore did some very impressive things with this.

If you're interested in learning more, look up David Cope, a researcher who has written several books on the topic. Or, you can see what Unreal's developers have been doing with procedural audio here: `http://proceduralaudionow.com/aaron-mcleran-and-dan-reynolds-procedural-audio-in-the-new-unreal-audio-engine/`. You can also find UE4 add-ons, such as a procedural MIDI plugin that I've worked with in the past.

Extending functionality with plugins and add-ons

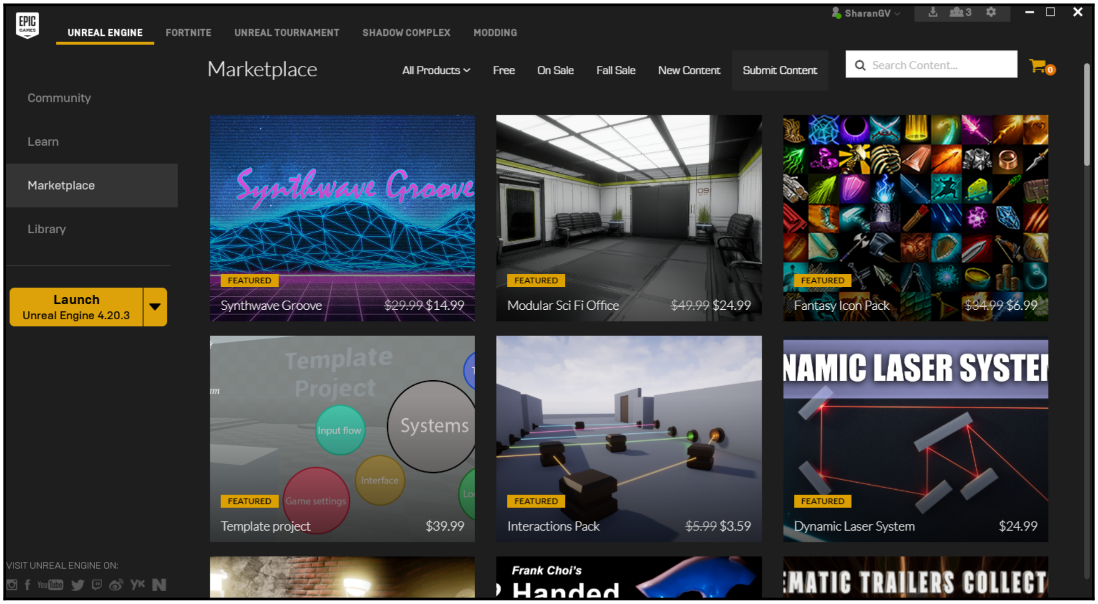
We've already seen a few examples of plugins and other add-ons, and how they can extend UE4, from adding VR functionality for your specific VR headset to adding functionality to support voxels, or procedural music. But there are a lot more available.

For plugins, you can go to **Edit** | **Plugins** and view everything that's already available by category:

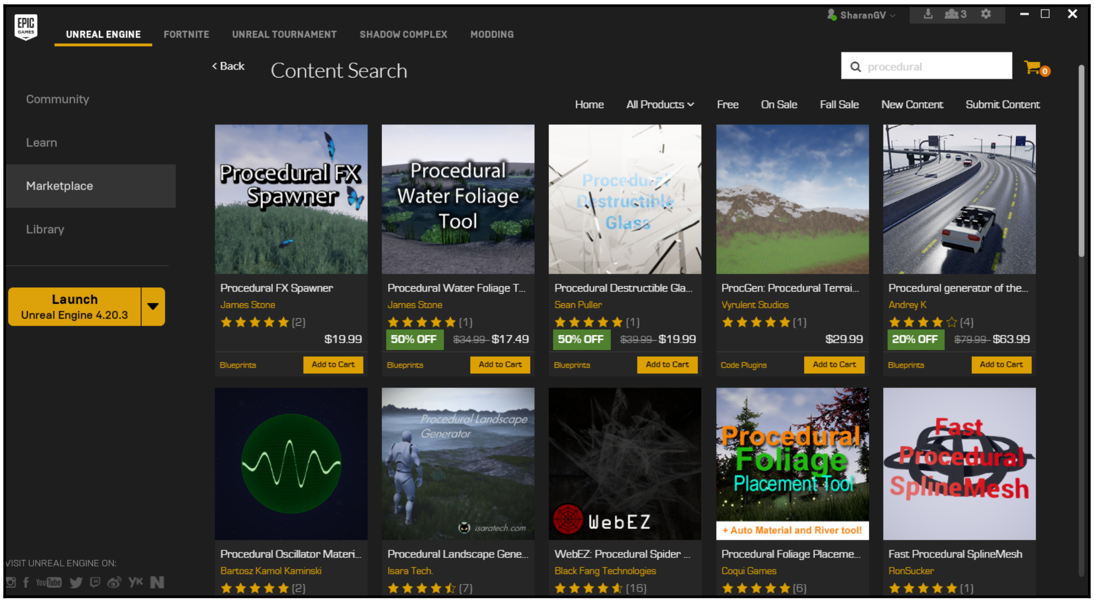


Those are the built-in plugins.

But if you're looking for more, you will want to check the **Marketplace** in the Epic Game's **Launcher**:



While a lot of what you'll see is graphics and models, there is plenty of functionality available that you can add on. Some of it is free, and some you need to pay for. For example, here's a search for **Procedural**:



UE4 is a very popular game engine, so if there's anything you need, there is a good chance that someone else has already developed an add-on for it. You can also find many projects elsewhere on the internet, many of which are open source with developers happy to help you out with their implementation. But these could take extra work to implement and you need to be careful and know exactly what you're downloading and installing.

Mobile, console, and other platforms

As you saw when we mentioned AR, you can develop in UE4 for mobile devices and preview your game either on your computer or on a phone. One of the great things about UE4 is that it supports many different platforms.

Many AAA game studios use UE4, so it definitely supports all the major game consoles (Xbox One, PS4, Switch, and even mobile consoles such as the 3DS and Vita). The trick to those is you can't usually just develop for them—you need to become an authorized developer and usually need to spend a lot of money on a DevKit (a specialized version of the console meant for development, that lets you debug on the console).

Fortunately, with the development of indie game marketplaces on the console, the bar to getting developer access is much lower now than it used to be. But you will still probably want a lot more experience and published game titles before you start looking into this.

Meanwhile, you still have many different options and platforms for your games. And once you've built a game for one platform, it's a lot easier to port that game to a different platform (UE4 makes that very easy!).

The main difference will be the controls, since you may be using a touchscreen, controller, motion controllers (in VR), or a keyboard and mouse. Each of these will have different requirements and will change gameplay slightly. But as long as you keep in mind which platforms you're aiming for from the very beginning, you will be able to plan your game in a way that will work for them all.

Summary

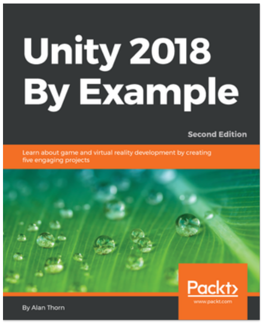
We've covered a lot in this book, but we've come to the end. We've learned the basics of C++, and created a really simple game in UE4 with some basic AI, a partial UI including inventory, and the ability to cast spells using particle systems. We also learned about VR, AR, and other up-and-coming new technologies that UE4 can help you out with.

You've now learned enough to start working on your own games. There are many other advanced books and websites you can look at if you need more information on specific topics, but you should have a much better idea of what you're looking at for now.

I hope you've enjoyed the journey. Good luck on your future projects!

Other Books You May Enjoy

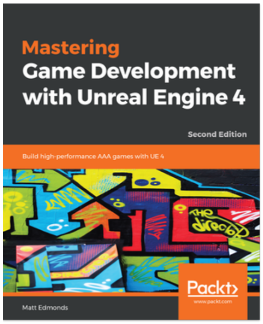
If you enjoyed this book, you may be interested in these other books by Packt:



Unity 2018 By Example - Second Edition
Alan Thorn

ISBN: 9781788398701

- Understand core Unity concepts, such as game objects, components, and scenes
- Study level-design techniques for building immersive and interesting worlds
- Make functional games with C# scripting
- Use the toolset creatively to build games with different themes and styles
- Handle player controls and input functionality
- Work with terrains and world-creation tools
- Get to grips with making both 2D and 3D games



Mastering Game Development with Unreal Engine 4 - Second Edition
Matt Edmonds

ISBN: 9781788991445

- The fundamentals of a combat-based game that will let you build and work all other systems from the core gameplay: the input, inventory, A.I. enemies, U.I., and audio
- Manage performance tools and branching shaders based on platform capabilities in the Material Editor
- Explore scene or level transitions and management strategies
- Improve visuals using UE4 systems such as Volumetric Lightmaps, Precomputed Lighting, and Cutscenes
- Implement audio-to-animation timelines and trigger them from visual FX
- Integrate Augmented Reality into a game with UE4's brand new ARKit and ARCore support
- Perform almost any game logic needed via Blueprint Visual Scripting, and know when to implement it in Blueprint as opposed to C++

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

T

U

V

W

X